Towards an
Object-Oriented Refinement Calculus

Jamie Barry Nathan Shield
BSc (Honours)

A thesis submitted to
The School of
Information Technology and Electrical Engineering
UNIVERSITY OF QUEENSLAND
for the degree of
DOCTOR OF PHILOSOPHY

5 February, 2004
Statement of Originality

I, Jamie Barry Nathan Shield, declare that the work presented in this thesis is, to the best of my knowledge and belief, original and my own work, except as acknowledged in the text, and that is has not been submitted, either in whole or in part, for a degree at this or any other university.
Acknowledgements

I would like to thank my supervisors, A/Prof David Carrington and Prof Ian Hayes, for being sources of information, inspiration, encouragement and friendship. Thankyou also to A/Prof Roger Duke for fulfilling the role of supervisor while David and Ian were on special study leave.

The Commonwealth of Australia supported me financially in the form of an Australian Postgraduate Award. The Software Verification Research Centre provided a conducive research environment, financed my attendance to a summer school, took me on as a research assistant and provided me with equipment. The School of Information Technology and Electrical Engineering financed my attendance at several conferences. Thankyou to the University of Southern Queensland for supporting me until completion.

A special thankyou to Helen and bub for their love and support.

I would also like to thank the following:

My family: Barry, Betty, Lindon, Cameron, Michelle, Nikita, Portia, Shakia, Bub, Cocky, Matthew, Seiko, and Miffy, and Helen’s family, including Nan, Pop, Peter, Denise, Danny, Rachel, Emily, Jessica, Harry, Thomas, Steven, Claire, Jack, Meg, Bub, Jamie, Ruth, Stephanie, and Meadow, for your emotional and financial support;

My colleagues, including Robert, Bradley, and Chris, who supplied technical support and stimulating discussions;

Bertha, Judy, Mark, Bimbo, Sooty, and Steve Waugh;

All of those people outside of work who kept me sane.
Abstract

This thesis is a step towards a programming method which supports a calculational style and is useful for the production of large software systems.

Object orientation is a popular programming paradigm for the development of large-scale systems. It is based on the idea that software should be a set of communicating entities, of which many entities are models of real objects.

The refinement calculus is a notation and set of rules that supports a calculational approach to the formal development of software.

It is envisaged that when the object-oriented paradigm and refinement calculus are fused, the resulting method would provide a development path for large-scale, provably correct software.

Several attempts have been made to construct an object-oriented refinement calculus. This thesis discusses several of these and summarises the remainder. The approaches include the modular reasoning work of Utting and Robinson [UR92], the type theoretic approaches of Mikhajlova and Sekerinski [MS97] and the higher-order approach of Cavalcanti and Naumann [CN99].

The thesis utilises a typed object calculus as the infrastructure of an alternative approach to the development of an object-oriented refinement calculus. Object calculi are a notation and set of rules that are used to model the concepts of object orientation. They have been used as the basis of several object-oriented languages.

Both a semantics of object values and a semantics of references for an object-oriented refinement calculus are presented within the thesis. Additionally, both object-based and class-based object-oriented refinement calculi are supported.

The development of programs that contain references can be tedious. A novel development method, termed coalescing, is introduced to allow unnecessary aliasing to be temporarily removed. Another novel technique is introduced to allow the development of reference semantics programs using a simpler value semantics.

Examples are presented which illustrate the capabilities of program development using an object-oriented refinement calculus. One example presents the application of the iterative, incremental development process of the object-oriented paradigm within the refinement calculus.
Contents

1 Introduction ................................. 1

2 Background ................................. 5
   2.1 Object Orientation ..................... 5
   2.2 Refinement Calculus ................... 9

3 Object-Oriented Refinement Calculus Approaches .... 17
   3.1 Modular Reasoning ..................... 19
   3.2 Sums of Products ....................... 23
   3.3 Storing Procedures in Variables ....... 26
   3.4 Miscellaneous Approaches ............. 27
   3.5 Conclusions .................... 29

4 A Typed Refinement Calculus .................. 31
   4.1 Object Theory and Calculi ............. 32
   4.2 Least-Upper and Greatest-Lower Bounds .. 40
   4.3 Predicate Transformer Foundation ....... 42
      4.3.1 State Modelling .................. 42
      4.3.2 Predicates ....................... 44
      4.3.3 Predicate Transformers .......... 46
      4.3.4 Statements ...................... 47
      4.3.5 Statement Refinements ........... 49
   4.4 Summary .............................. 50

5 An Object Representation .................... 51
   5.1 Motivation ............................. 51
   5.2 Additional Object Calculus Concepts .... 53
      5.2.1 Variance Annotations ............. 53
      5.2.2 Recursive Object Types .......... 54
      5.2.3 Dynamic Type Checking .......... 54
      5.2.4 Self Types ...................... 55
   5.3 Object-Typed Models .................. 56
   5.4 Summary .............................. 60

6 An Object-Oriented Language for Refinement ........ 62
   6.1 Client Constructs ..................... 63
      6.1.1 Field Selection .................. 64
      6.1.2 Object Specifications (Semantics for Values) .... 64
## CONTENTS

6.1.3 Field Update ........................................ 67
6.1.4 Method Invocation ................................... 68

6.2 Semantics for References ................................
6.2.1 Objects (Semantics for References) ................. 70
6.2.2 Client Constructs (Semantics for References) ...... 71

6.3 Summary .................................................. 77

7 Object- and Class-Refinement .............................. 78
7.1 Algorithmic Object-Refinement ........................... 79
7.2 Refinement of Client Constructs ......................... 82
7.2.1 Object-Assignment .................................. 83
7.2.2 Reference Cloning .................................. 91
7.2.3 Field Updates ....................................... 92
7.2.4 Introduce Method Calls .............................. 93

7.3 Private Attributes ....................................... 104
7.4 Data refinement for Objects ................................ 106
7.4.1 Simulation of Object-Data-Refinements ............ 111
7.5 Class-Based Refinement .................................. 118

7.6 Summary .................................................. 120

8 Towards A Reference Semantics ............................ 121
8.1 Aliasing Annotations .................................... 123
8.2 Reference Specifications ................................ 125
8.3 Data refinement via Inverse Commands ................. 128
8.4 Coalesced Programs ..................................... 130
8.4.1 Transforming to a Coalesced Program .............. 131
8.4.2 Transforming from a Coalesced Program ............ 132
8.4.3 Soundness of Coalesced Programming ............... 134

8.5 Semantics Conversion .................................... 135
8.5.1 Transforming to a Value Semantics ................ 136
8.5.2 Transforming to a Reference Semantics .............. 137
8.5.3 Proof of Semantic Conversions ...................... 138
8.5.4 Simulation .......................................... 138

8.6 Singly Linked List Example .............................. 140
8.7 Summary .................................................. 143

9 Object-Oriented Program Adaptation ...................... 144
9.1 Simultaneous Execution ................................ 144
9.2 Simultaneous Execution Example ....................... 148
9.3 Summary .................................................. 153

10 Conclusions ............................................... 154
10.1 Summary ................................................ 154
10.2 Results and Impacts ................................... 155
10.3 Future Work .......................................... 157
<table>
<thead>
<tr>
<th>A</th>
<th>Object Calculus</th>
<th>165</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.1</td>
<td>Axiomisation</td>
<td>165</td>
</tr>
<tr>
<td>A.2</td>
<td>Split Method Interpretation</td>
<td>172</td>
</tr>
<tr>
<td>A.3</td>
<td>Record Calculus</td>
<td>174</td>
</tr>
<tr>
<td>B</td>
<td>A Formal System of Objects</td>
<td>176</td>
</tr>
<tr>
<td>B.1</td>
<td>Predicate Calculus</td>
<td>176</td>
</tr>
<tr>
<td>B.2</td>
<td>Predicate Calculus Laws</td>
<td>177</td>
</tr>
<tr>
<td>B.3</td>
<td>Statements</td>
<td>179</td>
</tr>
<tr>
<td>B.4</td>
<td>Data Refinement Laws</td>
<td>188</td>
</tr>
<tr>
<td>B.5</td>
<td>Object-Refinement Laws</td>
<td>188</td>
</tr>
<tr>
<td>C</td>
<td>Lattice Theoretic Semantics</td>
<td>191</td>
</tr>
<tr>
<td>D</td>
<td>Proofs</td>
<td>193</td>
</tr>
<tr>
<td>D.1</td>
<td>State and Predicate Proofs</td>
<td>193</td>
</tr>
<tr>
<td>D.2</td>
<td>Predicate Transformer Proofs</td>
<td>195</td>
</tr>
<tr>
<td>D.3</td>
<td>Statement Proofs</td>
<td>196</td>
</tr>
<tr>
<td>D.4</td>
<td>Statement Refinements</td>
<td>199</td>
</tr>
<tr>
<td>D.5</td>
<td>Client Constructs Pre Object-Refinement</td>
<td>205</td>
</tr>
<tr>
<td>D.6</td>
<td>Object-Refinement Proofs</td>
<td>208</td>
</tr>
<tr>
<td>D.7</td>
<td>Client Constructs - Object-Refinement Proofs</td>
<td>211</td>
</tr>
<tr>
<td>D.8</td>
<td>Data refinement for Objects Proofs</td>
<td>226</td>
</tr>
<tr>
<td>D.9</td>
<td>Class Proofs</td>
<td>235</td>
</tr>
<tr>
<td>D.10</td>
<td>Semantics for References Proofs</td>
<td>237</td>
</tr>
<tr>
<td>D.11</td>
<td>Simultaneous Execution Statement Proofs</td>
<td>262</td>
</tr>
<tr>
<td>E</td>
<td>Symbol Glossary</td>
<td>264</td>
</tr>
<tr>
<td>F</td>
<td>Glossary</td>
<td>269</td>
</tr>
</tbody>
</table>
List of Figures

6.1 Attribute Path Closures ...................................................... 65
6.2 Object Specification Example ............................................ 73
7.1 Abstract Triangle ............................................................... 107
7.2 Concrete Triangle ............................................................. 108
8.1 Frame Annotations ............................................................. 125
8.2 Constrained Visualisation .................................................. 127
10.1 Introduce Subclass ............................................................ 158
C.1 Lattice Shapes ................................................................. 192
Chapter 1

Introduction

This thesis is a step towards a programming method in which designs and programs can be calculated from specifications and which supports the production of large software systems. The development of provably correct programs can be reliably accomplished for small programs via such methods as the “Refinement Calculus” [Mor94, MV94, BvW98]. The refinement calculus consists of a wide-spectrum language with a set of correctness preserving rules. The language is termed wide-spectrum as it extends Dijkstra’s [Dij76] guarded command language (GCL) with a non-executable ‘specification’ statement.

A program development in the refinement calculus is initiated by writing a specification statement which dictates the assumptions and desired effect of the program. Using rules provided by the refinement calculus, these specifications are systematically converted (or refined), with the aid of developer intuition, into the executable subset of the wide-spectrum language.

Unfortunately, the scalability of the refinement calculus is limited. Various efforts have been made to develop extensions to overcome the scalability constraints using approaches traditionally used to modularise (informal) program development: modules [Mor94], ADTs [Ban97], and object orientation [BMvW00, CN00, MS97, UR92]. This thesis focuses primarily upon the object-oriented approach, although the other approaches are not independent and many results of this thesis, especially the work on references, are pertinent to other approaches.

Object orientation is a programming paradigm centered around the idea that software should be a set of communicating objects, with each object encapsulating data plus the methods for manipulating that data. Object-oriented development, when performed correctly, can result in significant increases in code reuse and reliability. A thorough review of the object-oriented approach is given by Meyer [Mey97].

The motivation for fusing the object-oriented paradigm and the refinement calculus is that it is envisaged that the resulting method would provide a development path for the calculation of large-scale, provably correct software. Lano and Haughton [LH94] acknowledge that the application of formal specifications to large practical systems requires powerful system-structuring mechanisms: “a need which object orientation addresses ad-
mirably.” They suggest that an object-oriented language is not sufficient for the successful application of object-oriented techniques and that a disciplined approach is also required. The amalgamation of a refinement calculus with object orientation would produce such a disciplined method.

Despite previous work, a practical object-oriented refinement calculus does not yet exist. Researchers have, so far, been primarily concerned with the modelling of the core subtyping/dynamic dispatch features, with little attention being paid to the population of such calculi with practicalities. This thesis summarises the approaches previously considered, and presents an alternative approach, building on an object calculus which handles much of the required object-oriented infrastructure.

Object calculi have been extensively studied by Abadi and Cardelli [AC96] in an effort to bridge the gap between existing type theories and those required for object-oriented languages\(^1\). Object calculi are analogous to the \(\lambda\)-calculus [Bar81] except that they manipulate objects rather than functions. The study of object calculi has been instrumental in separating the concepts of inheritance and subtyping [CHC94]. The development of future object-oriented languages should benefit markedly from the utilisation of the key concepts of object calculi.

This thesis uses an object calculus as the infrastructure for both an object-based [SLU88] and class-based refinement calculus. This basis provides an abstract object representation. Current formulations of object-oriented refinement calculi expend significant effort dealing abstractly with the object-oriented semantics, rather than concentrating on the aspects of refinement. Goldsack and Kent [GK96, §2.3] identify that “the problems of combining object orientation and formal specification are due to the programming language origin of many of the concepts of object orientation.” For instance, object-oriented refinement has a close relationship with type compatibility [LH94, p78] [GK96, §2.4]. That is, if class \(A\) is a refinement of class \(B\), then it should be possible to use \(A\) as an instance of \(B\). As will be seen in Chapter 4, the direct support of object-oriented fundamentals in the object calculus (such as the determination of types and subtypes) allows for an abstract, concise expression of refinement relations. This indicates that an abstract object representation simplifies the development of an object-oriented refinement calculus. Forming an abstract object representation using an object calculus is thus a step in the right direction.

There are many object calculi upon which a refinement calculus can be built. Because the object calculus forms a foundation, the impact of the choice of object calculi is significant. This thesis has only trialled one such object calculus, namely Abadi and Cardelli’s \(\text{FOb}_{\mu}\). The constraints imposed by this particular choice are quite strong. One theoretical constraint is that unbounded nondeterminism is not supported. This constraint is discussed further in Chapter 5. Because of the impact of a particular object repre-

---

\(^1\)Palsberg and Schwartzbach [PS94, s2.6] discuss this gap and argue that its cause is due to the origins of type theory as a discipline of logic.
CHAPTER 1. INTRODUCTION

sentation, the thesis has built an abstraction of objects and used this as the basis for the object-oriented refinement calculus. Consequently, the proposed object representation can be easily replaced by another which does not have the aforementioned limitations. The only constraint is that the object representation must support embedded methods and the ability to add attributes to subtypes.

Goldsack and Kent \[GK96\] suggest that object-oriented structuring mechanisms can complicate reasoning about specifications. For instance, polymorphism makes it difficult to control which version of a method is actually executed, and the use of inheritance means that a method definition is fragmented across many classes. The integration of the object-oriented programming paradigm with the refinement calculus negates these concerns. Given a specification of a class \(A\), for class \(B\) to be a (type compatible) subclass of \(A\), each method in the subclass must be shown to be an incremental ‘improvement’, or refinement, of the corresponding method in the superclass. Consequently, it is irrelevant as to which method is invoked as all method variations conform to the original specification.

This thesis illustrates the use of several new development techniques. In particular, Chapter 8 introduces several techniques for easing the development of programs with a reference semantics.

This thesis is organised as follows. In Chapter 2 the preliminary concepts of object orientation and refinement calculi are introduced. Chapter 3 presents a summary of the existing approaches to object-oriented refinement calculi. Chapter 4 constructs a typed refinement calculus that supports a general rule of subsumption by appending type information to the classical refinement calculus definitions. To the author’s knowledge, this is the first presentation of a refinement relation that deals with the refinement of heterogeneously typed statements, i.e., the refinement of a statement of one type to a statement of a different type. Such refinements are useful when replacing an object with a sub-object within a client program. Heterogeneously typed refinement allows the client to use the new attributes of the sub-object. Chapter 5 identifies the requirements for an object model within an object-oriented refinement calculus built on the foundation of Chapter 4. A model is built that satisfies these requirements. In Chapter 6 a wide-spectrum, object-oriented language is developed. The notions of refinement and data refinement are presented in the language. A reference semantics and associated language constructs are also provided.

Chapter 7 introduces the notion of object-refinement—essentially an application of statement refinement to the method components of an object. The substitution of an object \(o\) in a program with an object-refinement is not a refinement as predicates such as \(o = c\) for some constant \(c\) are not upheld by the object-refinement. To achieve the desired object-refinement monotonicity in statements, predicates are restricted to those monotonic in object-refinement. This constraint has minor practical significance. The object-refinement relation is then generalised to an object-data-refinement relation. Finally, class-based extensions of these notions are provided.
Specialised concepts and techniques for developing reference semantics programs are presented in Chapter 8. In particular, a novel technique which supports the development of reference semantics programs using a simpler value semantics is introduced. Also a novel technique that removes unnecessary aliasing is presented.

Chapter 9 introduces a novel technique that involves the incremental enhancement of an object-oriented program using Groves’ [Gro98] simultaneous execution operator. This operator allows a new, enhanced specification to be given as a combination of the original program and the enhancements. This chapter also further illustrates the use of the object-oriented refinement calculus developed by the thesis.

Appendix A presents a complete axiomatisation of the object calculus used to construct the object-oriented refinement calculus. Appendix B provides a reference of additional laws, definitions and theorems.

Back and von Wright’s [BvW98] refinement calculus is founded in lattice theory, e.g., the refinement relation is a lattice ordering. Appendix C provides an introduction to lattice theory.

Appendix D presents the proofs of the theorems listed in the thesis. Appendix E provides a glossary for the symbol syntax. Finally, Appendix F provides concise definitions of the terminology used. It also acts as an index.

Throughout the thesis the nomenclature ‘theorem’ is used to denote a property whose proof has been given within the thesis, while ‘law’ is used to denote a property whose proof has been provided (and usually cited) elsewhere.
Chapter 2

Background

There is a diversity of terminologies used for object-oriented concepts. This chapter establishes the terminology and background concepts used within this thesis, including several core aspects of object orientation. Additionally, the foundations of refinement calculi are presented. The reader is also referred to Appendix C for an introduction to the lattice theory on which the refinement calculus is based.

2.1 Object Orientation

A review of object-oriented concepts is presented in this section. Meyer [Mey97] and Abadi and Cardelli [AC96] present introductions to the object-oriented programming paradigm.

The complexity of the problems that computers solve is rising. The solutions to these more complex problems demand languages with more abstract language constructs. The continuum between languages that do not even support abstract data structures through to object-oriented languages can be viewed in this context. Module-oriented languages offer encapsulation through abstract data structures (ADS). ADSs contain data and methods for manipulating this data. When an ADS is formulated, a single instance (or copy) is constructed. To acquire another copy another ADS must be formulated. For example, to model a car with four wheels, four ADSs would be required to model the wheels. The recognition of this inefficiency motivates the formation of abstract data types (ADT). An ADT contains data type information and methods for manipulating instances of the data type. Object-oriented languages are formed with the addition of the concepts of inheritance, polymorphism and dynamic dispatch.

Object Orientation “Object-oriented software construction is the building of software systems as structured collections of ... abstract data type implementations” [Mey97, p147]. The techniques for developing these structured collections are just as important, if not more, as the language. A proficient programmer can develop modularised code in a non-module-oriented language: the language merely serves to encourage good pro-
grammatical practice. Work on object-oriented techniques (such as UML [JBR99, PJ99] and Fusion [CAB+94]), heuristics [Rie96], and design patterns [GHJV95, Pre95] highlight the desire to capture and standardise object-oriented programming techniques. An object-oriented refinement calculus can be regarded in a similar vein: the formalisation of object-oriented techniques.

**Objects** If object orientation deals with implementations of ADTs, then objects are instantiations of ADTs. The term *fields* is used to denote the data portions of an object and the term *methods* is used to denote the procedural portions. *Attributes* refers to the combination of both. An attribute’s *host* is the object in which it is contained.

**Classes** A *class* can be viewed as a set of possible object instances or alternatively, as a template or blue-print, describing the construction of each object of the class. In addition, it can be considered a type, like an ADT [GK96].

**Subclasses** A *subclass* is a class constructed incrementally using another class. A class *C* can be extended with new attributes, or existing attributes can be overwritten to form a subclass (*D*). Class *C* is termed the *superclass* of *D*.

**Inheritance** is the sharing of attributes between a class and its subclass. Multiple inheritance occurs when a subclass inherits attributes from more than one class. Single inheritance occurs when a subclass inherits attributes from only one class.

**Object-based** The object-based approach [SLU88] supports the concept of objects but not inheritance [GK96, p7]. Object-based languages are intended to be both simpler and more flexible than class-based languages. They typically use special objects termed *prototypes* as the templates for the creation of object instances. *Cloning* is the term that refers to the process of copying an object.

**Polymorphism** Given classes *C* and *D* where *D* is a subclass of *C*, then an instance of class *D* (*d*) may be used as an instance of class *C*. This is referred to as (subtyping) *polymorphism*.

**Subtyping** Traditionally, the polymorphic reuse of objects is restricted to situations of subclassing. There are occasions, however, when it is desirable to use an object (*d*) as an instance of a differing class (*C*) even though its class (*D*) may not be a subclass (of *C*). *Syntactic subtyping* is introduced for this reason. It allows *polymorphic* reuse of objects regardless of whether their respective classes exist in a subclass relationship. This increase in flexibility is part of the approach known as inheritance-is-not-subtyping [CHC94].
An object’s (and class’s) type is determined by a list of the names of its attributes and their types (types of its fields and the parameters of its methods). One possible definition of the subtyping relationship is that subtypes may introduce new attributes while maintaining the old. With this definition, a subtype has at least the attributes of its supertype. For object types $\gamma$ and $\delta$, if $\delta$ has at least the attributes of $\gamma$, then $\delta$ is a subtype of the supertype $\gamma$:

$$\delta \preceq \gamma$$

Consequently, given a class $C$ with type $\gamma$, and another class $D$ of type $\delta$, then code written to use an instance of class $C$ would also operate on an instance of class $D$. The effect of the code may be substantially different despite this syntactic compliance.

**Subsumption** Given types $\gamma$ and $\delta$ where $\delta \preceq \gamma$, and an object $d$ of type $\delta$, then the recognition that $d$ is also of type $\gamma$ is known as subsumption.

$$d : \delta \wedge \delta \preceq \gamma \Rightarrow d : \gamma$$

**Dynamic Dispatch** Given class $C$ with method $m$ and subclass $D$, when method $m$ is invoked on an instance $d$ of class $D$ (and by subsumption, also an instance of $C$), it is $D$’s version of the method $m$ that is executed, not $C$’s. This is termed dynamic dispatch. It has several synonyms including dynamic binding and runtime method discrimination.

**Subclassing-is-not-Subtyping** As discussed above, allowing subtyping in the absence of subclassing provides more opportunities for reuse of objects. The alternative should also be considered: subclassing without subtyping. The following example, adapted from one of Abadi and Cardelli’s [AC96, p31], shows that subclassing does not necessarily induce subtyping.

**Example 2.1 (Subclassing-is-not-Subtyping)** The class $maxClass$ has a field $n$ of type $Z$ and a method $max$ which takes another instance of class $maxClass$, and returns the object with the largest $n$ field. The syntax used in this example is informal.

```plaintext
class maxClass is
  var n : Z := 0;
  method max(other : Self) : Self is
    if (self.n > other.n) return self else return other end
  end
end
```

The subclass $minMaxClass$ includes an additional method for returning the object with
the minimum $n$.

```plaintext
subclass minMaxClass of maxClass is
  method min(other : Self) : Self is
    if (self.n < other.n) then return self else return other end
  end
end
```

The object type of the first class `maxClass` is:

```plaintext
ObjectType MaxType is
  var n : Z;
  method max(other : MaxType) : MaxType
end
```

Correspondingly, the object type of class `minMaxClass` is

```plaintext
ObjectType MinMaxType is
  var n : Z;
  method max(other : MinMaxType) : MinMaxType
  method min(other : MinMaxType) : MinMaxType
end
```

Under the subtyping definition these types are not related as the method parameter types are not equivalent\(^1\).

Now consider a subclass of `minMaxClass`, namely `newMinMaxClass`. It overrides the `max` method to use the minimum method.

```plaintext
subclass newMinMaxClass of minMaxClass is
  override max(other : Self) : Self is
    if (other.min(self) = other) then return self else return other end
  end
end
```

An instance `mm` of `newMinMaxClass` has the type `newMinMaxType`.

```plaintext
ObjectType newMinMaxType is
  var n : Z;
  method max(other : newMinMaxType) : newMinMaxType;
  method min(other : newMinMaxType) : newMinMaxType
end
```

Since the label `newMinMaxType` acts like a bounded quantifier it can be replaced with another label, e.g., `MinMaxType`. `MinMaxType` and `newMinMaxType` are therefore the same type. Consequently the instance `mm` of `newMinMaxClass` has type `MinMaxType`.

\( mm : MinMaxType \)

---

\(^1\)This constraint is relaxed later.
This is referred to as structural type equivalence. Name type equivalence is structural type equivalence with an additional constraint that the names are also equal.

Assuming the argument that subclassing implies subtyping means that subtyping should hold between MinMaxType and MaxType.

\[(\text{minMaxClass}\ \text{subclass}\ \text{maxClass}) \implies \text{MinMaxType} \preceq \text{MaxType}\]

Consequently, through subsumption, \(mm : \text{MaxType}\).

\[(mm : \text{MinMaxType} \land \text{MinMaxType} \preceq \text{MaxType}) \implies mm : \text{MaxType}\]

Given object instance \(m\) of \(\text{maxClass}\) and instance \(mm\) of \(\text{newMinMaxClass}\), then the call \(mm.\text{max}(m)\) would produce a dynamic error as \(m\) does not possess a \(\text{min}\) method.

\[
\begin{align*}
mm.\text{max}(m) \\
& \equiv \text{Argument substitution.} \\
& \text{if } (m.\text{min}(mm) = m) \text{ then return } mm \text{ else return } m \text{ end}
\end{align*}
\]

Given this contradiction, it must be deduced that subclassing does not always imply subtyping. The reader is referred to the work of Cook, Hill and Canning [CHC94] for a thorough discussion of this topic.

\[\Diamond\]

### 2.2 Refinement Calculus

This thesis is a step towards a programming method in which designs and programs can be calculated from specifications and which supports the production of large software systems. The object-oriented concepts presented in the previous section are used to help provide support for large software systems. The refinement calculus concepts presented in this section are used to support the calculational aspects of the thesis.

The refinement calculus is a notation and set of rules for the calculation of executable programs from specifications [Bac80, Mor87, Mor94, BvW98]. One inspiration for the refinement calculus was Dijkstra’s [Dij76] weakest precondition semantics. His work provides a predicate transformer semantics for the guarded command language and also lists several properties, termed healthiness conditions, that programs written in the guarded command language possess. The work on the refinement calculus showed that some of these conditions are overly restrictive and that by weakening them it is possible to include interesting new constructs in the language. The main addition to the language is the specification statement. Adding the specification statement to the guarded command language means that the newly representable abstract programs no longer obey Dijkstra’s Law of the excluded miracle—thereby allowing the production of miraculous, non-implementable programs, if the developer is not careful.
The inclusion of specifications in the same language as the implementation or code constructs leads to a language described as wide-spectrum and distinguishes the refinement calculus from earlier work because derivations of programs can now be carried out within a single semantics.

Dijkstra’s weakest precondition semantics presented language constructs as functions: the construct

\[ wp(P, post) \]

returns the weakest precondition necessary for the program \( P \) to execute and terminate in a state satisfying the postcondition \( post \). Specifications can be written in this style:

\[ pre \Rightarrow wp(P, post) \]

meaning “if activated in a state for which \( pre \) holds, the program \( P \) must terminate in a state for which \( post \) holds.”

Morgan’s version of the refinement calculus mutated such specifications to the following form under the assumption that it will be transformed into \( P \):

\[ [pre, post] \]

This new form is considered an (abstract) statement. Morgan’s weakest precondition semantics of this statement is:

\[ wp([pre, post], R) \triangleq pre \land (\forall \bar{v} \bullet post \Rightarrow R) \]

That is, for the specification to terminate in a state satisfying \( R \), in the initial state \( pre \) must hold and essentially, \( post \) must be a stronger condition than \( R \). The specification may terminate in a variety of states all of which satisfy \( R \). A quantification of the program variables \( \bar{v} \) over \( post \Rightarrow R \) is used to strengthen the weakest precondition predicate to handle all of these states. This specification statement has been both improved and decomposed in various ways by Back and Morgan. For instance, Morgan presents a framed specification which constrains the variables that may be altered. The frame \( w \) identifies the modifiable variables.

\[ wp(\bar{w}; [pre, post], R) \triangleq pre \land (\forall \bar{w} \bullet post \Rightarrow R)[\bar{v}_0 \backslash \bar{v}] \]

Morgan allows the convention that zero-subscripted variables in \( post \) denote the initial state of those variables. Consequently the substitution, after quantification of the ‘final’ variables, produces a predicate on the initial state.

Given an abstract program including specifications, the goal is to produce an executable program through a process of refinement. The program \( P \) refines to the program \( Q \), written \( P \sqsubseteq Q \), if for all (postcondition) predicates \( (R) \) the weakest precondition of \( Q \)
with respect to that predicate \( R \), denoted by \( wp(Q, R) \), is weaker than (or will be guaranteed by) the corresponding weakest precondition of \( P \):

\[
P \sqsubseteq Q \iff \forall R \bullet wp(P, R) \Rightarrow wp(Q, R)
\]

Alternatively, for any postcondition \( R \), if \( P \) achieves \( R \) from a particular initial state, then \( Q \) will also achieve \( R \) from that same state.

The most important property of the refinement calculus is monotonicity. This allows individual program fragments to be refined and then substituted back into the program with the result being that the entire program is refined. Given programs \( F(P) \) and \( F(Q) \) then if \( P \sqsubseteq Q \) then

\[
F(P) \sqsubseteq F(Q)
\]

Consequently, programs can be refined piece-wise. The statements in the guarded command language are all monotonic with respect to refinement.

Using the refinement relation, along with the weakest precondition semantics of the specification statement and the classical imperative language constructs (as given by Dijkstra), various refinement rules can be provided. Once these rules have been developed, the underlying weakest precondition semantics can be ignored, thereby providing a method that allows the calculation of programs from initial specifications. Typically this process involves some intuition to guide the development process. This process is termed algorithmic refinement, or procedural refinement.

The remainder of this section introduces data refinement, both in general and in the context of the refinement calculus [Bac88, Mor89, BvW90, Mor90, MG90, vW92, vW94, GM93]. Also presented is the technique of simulation [Mil71] which ‘wraps up’ the use of data refinement as an algorithmic refinement so that data refinements can be used in practice. These concepts are used in Chapter 8 for temporarily data refining a reference semantics program into a simpler value semantics to allow for easier algorithmic refinements.

An abstract data type is a piece of data and a set of operations that have exclusive access to view or modify the piece of data. Informally, data refinement denotes the replacement of a data type within a program to increase efficiency and/or produce implementable code. Typically, an abstract, mathematically clear and concise data type (termed here a specification) is replaced with a more concrete, implementation-like data type. The replacement of the specification data type with the implementation data type in client code is referred to as simulation. Data refinement, then, is a formal, mathematical relationship between program fragments; namely the respective operations of the specification and implementation data types.

Given the data refinement of a specification by an implementation data type, simulation is the proof of refinement between the specification client code and the implementation client code. The behaviour of the specification client code has been ‘simulated’ by the implementation client code—even though they work on different state spaces.
Data Refinement Examples  There are three types of relationship that may exist between specification and implementation states. Two of these are functional relationships and the third is relational. One of the functional relationships occurs when the specification state is a function of the implementation state. This can be illustrated, for instance, when implementing a set as a sequence. The set

\{1, 2, 3\}

can be represented as the sequence

\langle 1, 2, 3 \rangle

or, alternatively, as

\langle 3, 1, 2 \rangle

Given \( n \) as the cardinality of the specification set, there are \( n! \) implementation representations of the specification state as a sequence with no duplicates. Morgan [Mor94] provides specialised rules for data refinements of this type. In this example, the implementation state has extra (positional) information that was included as a side effect of the data refinement of the set by a sequence. This information is not needed and hence may be ignored.

Another relationship occurs when the implementation state is a function of the specification. An example of this occurs for programs that average a list of numbers [Mor94, p170]. Given a specification data representation of a sequence of numbers, e.g., \langle 3, 4, 5 \rangle, data refinement could be used to change the representation to a pair containing the sum and length of the sequence: \( (12, 3) \). The average can be determined using either data structure. For this type of data refinement, the specification state has more information than is required. The data refinement removes the unneeded information.

The third, relational, data refinement relationship is illustrated in the following example.

Example 2.2 (Variable List) A program maintains a list of variable names and their associated string values. As variables are declared they are appended to a sequence. Given the variables \( a \) and \( b \) with values \textit{dog} and \textit{cat} respectively,

\langle (a, \textit{dog}), (b, \textit{cat}) \rangle

declaring a new variable \( c \) with value \textit{horse} generates the sequence:

\langle (a, \textit{dog}), (b, \textit{cat}), (c, \textit{horse}) \rangle

For efficiency reasons, the designer decided the list should be able to dynamically reconfigure itself such that the most frequently accessed variables are at the head of the list, thereby decreasing access time. To achieve this, the declaration-ordered sequence is
data refined to a sequence of triples ordered on the third triple element—which indicates the number of variable accesses. Given a variable $a$, accessed six times and $b$, accessed eight times, the sequences would be:

$$\{(b, \text{cat}, 8), (a, \text{dog}, 6)\}$$

Declaring a variable $c$ would result with:

$$\{(b, \text{cat}, 8), (a, \text{dog}, 6), (c, \text{horse}, 0)\}$$

No functional relationship exists between the specification and implementation data structures. The specification data structure

$$\{(a, \text{dog}), (b, \text{cat})\}$$

could be represented by an implementation data structure in which $a$ is accessed more:

$$\{(a, \text{dog}, 7), (b, \text{cat}, 4)\}$$

or alternatively an implementation data structure in which $b$ is accessed more.

$$\{(b, \text{cat}, 8), (a, \text{dog}, 6)\}$$

Conversely, the implementation data structure

$$\{(b, \text{cat}, 8), (a, \text{dog}, 6)\}$$

could be representative of a specification data structure in which $b$ was declared first:

$$\{(b, \text{cat}), (a, \text{dog})\}$$

or, by a data structure in which $a$ was declared first.

$$\{(a, \text{dog}), (b, \text{cat})\}$$

The specification data structure has additional declaration-order information that was stored as a side effect\(^2\). The implementation data structure loses this information yet gains extra ‘access information’ that is used for efficiency gains. In this example a functional relationship exists between the specification and implementation data structures once the extra, unneeded information of the specification data structure is removed. That is;

$$f(\text{specification}) = g(\text{implementation})$$

Here $f$ is simply the range of the sequences, indicating that the ideal data structure is a set of pairs relating variables with their values. The function $g$ removes the third triple element and the ordering information.

$$g(\text{implementation}) = \{el \in \text{ran implementation} \bullet (\text{fst}(el), \text{snd}(el))\}$$

\(^2\)This extra information is termed implementation bias [dREB\(^+\)98, p8].
Here the specification data structure
\[ \{(a, \text{dog}), (b, \text{cat})\} \]
is mapped through \( f \) to
\[ \{(a, \text{dog}), (b, \text{cat})\} \]
as is the implementation data structure (through \( g \))
\[ \{(b, \text{cat}, 8), (a, \text{dog}, 6)\} \]

One approach to data refinement, as presented by Morgan et al. [MV94], is to introduce a predicate transformer (\( rep \)) that provides a link between specification and implementation predicates (as shown in Definition 2.3).

**Definition 2.3 (Data Refinement)** The specification \( Prog_s \) data refines to the implementation \( Prog_i \) under \( rep \) (\( \preceq_{rep} \)) if:

\[ rep; \ Prog_s \preceq Prog_i; \ rep \]

That is:

\[ Prog_s \preceq_{rep} Prog_i \rightleftharpoons rep; \ Prog_s \preceq Prog_i; \ rep \]

Rules have been developed that allow \( rep \) to be ‘pushed through’ each language construct, piece-wise converting them to an appropriate implementation. For example, laws 2.4 and 2.5 are data refinement rules for specifications and sequential composition respectively. The nomenclature ‘law’ is used in this thesis to denote properties that have been proven by other researchers and which are merely listed. The nomenclature ‘theorem’ is used to denote properties that have been proven in this thesis.

**Law 2.4 (Data Refine Specification)** For the data refinement of specifications, \( \alpha \) denotes the specification variables being removed, \( \beta \) the implementation variables being introduced and \( \gamma \) the variables common to both specification and implementation states.

Morgan and Gardiner [MG90, p95, Corollary 1] introduce a (validity) data refinement rule for specifications in which the postcondition \( post \) does not contain initial variables. For subset \( \delta \) of the abstract variables \( \alpha \), i.e., \( \delta \subseteq \alpha \), and \( rep \ p \rightleftharpoons (\exists \alpha \cdot AI \land p) \) where \( AI \) is the abstraction invariant:

\[ \delta, \gamma; [pre, post] \]
\[ \preceq_{rep} \]
\[ [\begin{array}{l}
\varepsilon_{\alpha} \\
\beta, \gamma; [AI \land pre, (\exists \delta \cdot AI \land post)]
\end{array}] \]
The syntax $[[\mathtt{con} \ \alpha \ \bullet ...]]$ is used to introduce the variables $\alpha$ as logical constants. Logical constants are magically chosen values that are not code. The reader is referred to page 184 in Appendix B.3 for the definition.

Morgan and Gardiner show that data refinement distributes through logical constants [GM91, p80, Lemma 8]. Consequently, the data refinement rule can be generalised to allow initial variables in $post$.

$$\delta, \gamma: [\mathtt{pre}, \mathtt{post}]$$

$\equiv$ Initial Variable (B.49)

$$[[\mathtt{con} \ D, G \ \bullet$$

$$\delta, \gamma: [\mathtt{pre} \land D = \delta \land G = \gamma, \mathtt{post}[\delta_0, \gamma_0\setminus D, G]]$$

$$]] \preceq_{\text{rep}}$$

$$[[\mathtt{con} \ D, G, \alpha \ \bullet$$

$$\beta, \gamma: [\mathtt{AI} \land \mathtt{pre} \land D = \delta \land G = \gamma, (\exists \delta \bullet \mathtt{AI} \land \mathtt{post}[\delta_0, \gamma_0\setminus D, G])]$$

$$]]$$

This rule also deals with assignments via Theorem B.52.

The following rule is used for the piece-wise data refinement of sequential compositions, given the data refinements of each construct.

**Law 2.5 (Data refine Sequential Composition)** Given $pt_1 \preceq_{\text{rep}} pt'_1$ and $pt_2 \preceq_{\text{rep}} pt'_2$

$$pt_1; \ pt_2 \preceq_{\text{rep}} pt'_1; \ pt'_2$$

The piece-wise style is exemplified by the proof of this law.

$$\preceq \bullet \text{Using the assumption } pt_1 \preceq_{\text{rep}} pt'_1$$

$$pt'_1; \ \preceq \bullet \text{Using the assumption } pt_2 \preceq_{\text{rep}} pt'_2$$

$$\preceq \text{rep}$$

The disadvantage of using such an approach is that the chosen $\text{rep}$ must be shown to possess the following properties:

**Desideratum 2.6 (Universal Join Homomorphism of rep)** For any (specification) predicates $\phi_i$,

$$\text{rep}(\bigvee_i \phi_i) \equiv \bigvee_i (\text{rep}(\phi_i))$$

Gardiner and Morgan [GM91] refer to this property as $\lor$-distributivity.

From this property the following two properties hold. They are listed in their own right for ease of reference.
Desideratum 2.7 (Strictness of rep) Strictness (also known as feasibility) denotes that the only way to achieve the impossible is to start with the impossible. Strictness is required, for example, to allow abort to data refine to itself.

\[ \text{rep}(\text{false}) \equiv \text{false} \]

This is a special case of Desideratum 2.6 where the disjunction is empty: \( \bigvee_{i \in \emptyset} \phi_i \equiv \text{false} \).

Desideratum 2.8 (Monotonicity of rep) rep must also be monotonic. That is, given \( p \Rightarrow q \), then \( \text{rep} \ p \Rightarrow \text{rep} \ q \) holds. The proof that join homomorphism implies monotonicity has been completed by Back and von Wright [BvW98, p45].

Finally, the prepending of rep to the specification program, and its removal from the implementation program, requires explanation. This is part of the process of simulation.

Simulation An early definition of simulation involved defining a relation between two programs which were “said to be realizations of the same algorithm” [Mil71]. This was used by Hoare [Hoa72, HHS87] and provided the basis for data refinement in the refinement calculi. More recently, de Roever et al. [dREB+98] have summarised data refinement and simulation techniques used in several areas of specification and refinement, including Z, VDM, and Back’s refinement calculus.

Morgan and Gardiner [MG90, p485] present a “soundness of data refinement” theorem which allows simulation in the context of data refinement. After calculating the corresponding implementation program fragment, the specification variable \( (s : S) \) block (with initialisation Init) can be refined by the implementation variable \( (i : I) \) block (with initialisation rep Init), i.e.,

\[
\begin{align*}
\left\llbracket \text{var } s : S \mid \text{Init} \bullet \text{pt} \right\rrbracket & \equiv \\
\left\llbracket \text{var } i : I \mid \text{rep Init} \bullet \text{pt'} \right\rrbracket & \\
\end{align*}
\]

provided \( \text{pt} \preceq_{\text{rep}} \text{pt'} \). Thus the data refinement of the specification code forms part of the proof of the ‘simulation’ of the specification variable block by the implementation variable block.
This chapter provides an introduction to the modelling of object-oriented refinement calculi. The description of issues is followed by summaries of several object-oriented refinement calculi.

There have been several attempts at producing an object-oriented refinement calculus, and all vary significantly in both their foundations and features. In particular, the representation of objects varies significantly amongst the calculi. Classical object-oriented languages portray the inclusion of fields and methods in objects as desirable. However, encapsulated methods are difficult to represent due to their recursive nature: methods manipulate the objects of which they are a component. Utting [Utt92] presents this problem in the context of denotational semantics and proves that, in general, there is no solution to the domain equations that allow fields and methods in the same object. Naumann [Nau94b, p471] however, specialises the general case and show that for certain object-oriented programs, a solution can be found.

When developing a semantics for an object-oriented refinement calculus, the binary method problem [BCC+95] is relevant for type-theoretic attempts. Methods which use a value argument of the same type as the method’s host (termed the self type) and return a result argument, also of the self type, are called binary methods. Type theoretic researchers have had difficulty typing objects that contain binary methods as they contain both covariant and contravariant\(^1\) occurrences of the self type [AC96, p209]. Unfortunately, an object-oriented refinement calculus semantics has a need for such objects. An intuitive, abstract representation of objects in an object-oriented refinement calculus is one that contains methods which are predicate transformers that manipulate the host object. That is, the predicate transformers are functions from predicates on a state type involving the host type to predicates on a state type also involving the host type. Consequently these predicate transformers have similar problems to binary methods.

The most common solution to the binary method problem has been to decouple fields

\(^1\)Defined on page 20.
and methods. Typically this involves using an object comprised of fields and an external collection of methods that manipulate the object’s fields. This however, violates the encapsulation principles of object orientation—though (through practical restrictions) only in the semantics.

In the context of an object-oriented refinement calculus, the models of object-oriented concepts such as object representation, dynamic dispatch and field update are highly dependent upon each other.

The modelling of dynamic dispatch falls into one of two approaches, based upon whether the object representation decouples methods. If methods are embedded then dynamic dispatch can be modelled by the execution of the contained method. When methods are not embedded in objects a mechanism must exist for associating them. The approach of Mikhajlova and Sekerinski [MS97] uses an implicit tag whereby object instances are linked to their declaration class which contains a list of methods. Utting [Utt92] provides a function that, in its simplest form, maps object identities, through method names, to the method to be invoked.

A common aspect of object-oriented refinement calculi is their treatment of behavioural conformance of polymorphic objects: given classes \( C \) and \( D \) where \( D \) is a subclass of \( C \), and a program \( S \) written for instances of class \( C \), then using subsumed instances of class \( D \) instead produces a refinement. This is the fundamental property of (class-based) object-oriented refinement calculi. If this constraint was not applied then it would be possible to form an arbitrary subclass. Instances of this subclass could be subsumed and used as instances of the superclass. The behaviour of such instances would not necessarily conform to that dictated by the superclass specification. By fusing polymorphic reuse with behavioural conformance correct class behaviours can be guaranteed.

The manner in which behavioural conformance of polymorphic objects, also known as modular reasoning [Utt92], is implemented varies significantly among object-oriented refinement calculi, and will be examined on a case-by-case basis.

At present most research in the field of object-oriented refinement calculi has concentrated on the modelling of object-oriented concepts with little attention paid to the development of additional refinement techniques that an object-oriented refinement calculus allows.

The remainder of the chapter is a collection of summaries of existing object-oriented refinement calculi. Each summary loosely follows a format consisting of theoretical foundations, object representation, dynamic dispatch, field selection and update, subtyping, class representation, and behavioural conformance. The syntaxes of the various calculi have been modified for readability and consistency.
3.1 Modular Reasoning

Utting and Robinson [UR92] use a lattice-theoretic framework. They do not propose a specific object representation. They assume, however, that objects are a subset of the value space, and that methods are not embedded in objects.

To model the object’s methods and consequently dynamic dispatch, they define a ‘late binding’ function

$$\psi : P \rightarrow Val^* \rightarrow Prog$$

that maps procedure names ($P$) to functions ($Val^* \rightarrow Prog$). The functions map finite sequences of values ($Val^*$) to program code ($Prog$). The values are used to determine the code to execute. For single dispatch, object-oriented languages, the (single) value used would be the host object. However, since multiple values (objects or method arguments) can be used to determine the code to execute, the resulting language is multiple dispatch.

For multiple dispatch languages the dynamic dispatch mechanism is not dependent upon a single object instance. In this context the decoupling of methods and fields is natural. However, Utting and Robinson indicate that this multiple dispatch mechanism may permit inconsistencies and a loss of object encapsulation as the object(s) used to determine dynamic dispatch may be different from the object which the method manipulates. Consequently, they enforce links between parameterisation and dynamic dispatch. That is, they ensure the values (objects) used to determine the code to execute are the same objects that the selected code uses.

Utting and Robinson constrain a subtyping relation with behavioural conformance constraints for polymorphic objects. This modular reasoning is achieved by restricting their late binding function ($\psi$) to be monotonic with respect to the subtyping relation. For objects $c$ and $d$, where $d$ is a subtype of $c$ ($d \preceq c$), and where both have a method $p$, then the substitution of $c$ by $d$ in a program is a refinement as the monotonicity of $\psi$ guarantees that

$$\psi p c \sqsubseteq \psi p d$$

In comparison, other object-oriented refinement calculi achieve comparable properties by restricting the subclass relation with behavioural conformance constraints, e.g., refinement. By also restricting subsumption to the class hierarchy, similar monotonic properties can be deduced.

Utting and Robinson weaken the monotonicity requirement by using static properties known about arguments passed to the invoked methods. This allows increased flexibility while maintaining modular reasoning. It also means that transitivity is lost. To regain transitivity, a transitive modular reasoning relation is provided, thereby easing proof of modular reasoning for a chain of subtypes. Unfortunately, the subtyping relation does not fully support data refinement. Utting and Robinson have commenced the development
of a more general relation but proof that it maintains the properties of transitive modular reasoning is not complete. Another problem, suggested by Sekerinski [Sek96, p14], is that technical reasons preclude specifications within the methods of Utting and Robinson.

To provide a comparison between both object-oriented refinement calculi presented in this chapter, each approach is applied to a simple example specification (and/or refinement) of a bag. The following example presents the refinement of a bag using modular reasoning and the construction of an associated late binding function.

**Example 3.1 (Refinement of Bag using Modular Reasoning)**

Consider a bounded bag object with a `bagdata` field representing the bag information, and a `bound` field containing the maximum number of elements allowed in the bag. The cardinality of the bag is constrained to be not greater than the bound: \( \#(a \circ bagdata) \leq a \circ bound \). The \# symbol is bag size which is inclusive of frequencies. That is, duplicates in the bag are counted. The bounded bag object also contains a `put` method which, if possible, inserts a new element into the bag, a `get` method which returns an element from the bag (removing it at the same time) and a `card` method for determining the cardinality of the bag. Such an object is illustrated by the following informal syntax where `Bound` is some given constant, and `PUT`, `GET` and `CARD` are the respective method bodies:

```plaintext
object
    private field bagdata : bag(ELEMENTS) := []
    private field bound : \mathbb{N} := Bound
method put(value el : ELEMENTS) = PUT
method get(result el : ELEMENTS) = GET
method card(result num : \mathbb{N}) = CARD
end
```

where `bag(\text{ELEMENTS})` is a bag of elements.

The only constraint Utting and Robinson place on their object representation is that objects `Obj` are a subset of the value space `Val`.

\[ \text{Obj} \subseteq \text{Val} \]

Their examples, however, define objects using partial functions from a set of names `Var` to values. This example uses this same representation.

\[ \text{Obj} \cong \text{Var} \rightarrow \text{Val} \]

Subtyping (\( \preceq \)) is defined within the examples of Utting and Robinson so that objects with a set of attributes are subtypes of those objects with a subset of those attributes. Additionally, the fields must exist in a subtyping relationship.

\[ a \preceq b \iff a \in \text{Obj} \land b \in \text{Obj} \land \text{dom } b \subseteq \text{dom } a \land \forall i : \text{dom } b \bullet a \circ i \preceq b \circ i \]

The record field types are said to vary covariantly. **Covariance** is the monotonicity property of record field types according to the subtype relation. If the field types vary anti-monotonically, they are said to be **contravariant**. For simplicity, it is assumed that basic,
non-object types subtype discretely (only reflexively), i.e., type \( A \) is a subtype of type \( B \) \( (A \preceq B) \) if and only if \( A \) is \( B^2 \).

The parameterised predicate \( \text{BagType} \) captures the required typing properties of the fields of a bounded bag.

\[
\text{BagType}(a) \triangleq a \in \text{Obj} \land \text{dom} a = \{\text{bound}, \text{bagdata}\} \land \ a \triangleq \text{bound} \in \mathbb{N} \land a \triangleq \text{bagdata} \in \text{bag}(\text{ELEMENTS})
\]

These constraints are that \( a \) is an object \( (a \in \text{Obj}) \) with fields \( \text{bound} \) and \( \text{bagdata} \) \( (\text{dom} a = \{\text{bound}, \text{bagdata}\}) \); that \( \text{bound} \) is a natural number and that the bag must consist of elements of the given set \( \text{ELEMENTS} \). The parameterised predicate \( \text{Bag} \) captures all desired properties of the bounded bag, typing and otherwise.

\[
\text{Bag}(a) \triangleq \text{BagType}(a) \land \#(a \triangleq \text{bagdata}) \leq a \triangleq \text{bound}
\]

The second conjunct constrains the cardinality of the bag to be not greater than the bound.

Unfortunately the predicate \( \text{BagType} \) is not applicable to subtypes of \( \text{Bag} \). For instance, subtypes may wish to introduce new fields, yet \( \text{BagType} \) constrains the fields to be \( \text{bound} \) and \( \text{bagdata} \) only. Monotonic closure is used to generalise the \( \text{BagType} \) predicate for subtypes.

**Definition 3.2 (Monotonic Closure)** Monotonic closure is defined as:

\[
\bar{P} \triangleq \exists u'. P[u\backslash u'] \land u \preceq u'
\]

where \( P \) is a predicate, \( u \) is a sequence of distinct variables (the subtyping relationship here applies pointwise to the elements of the sequences). A starred notation, \( \bar{P} \), is introduced to denote the monotonic closure with respect to all names, \( \text{Var} \).

The only effect of using the monotonic closure of \( \text{BagType}(\bar{\text{BagType}}(\text{arg})) \) is that the predicate constraining \( a \)'s domain is generalised to a superset of \( \{\text{bound}, \text{bagdata}\} \):

\[
\text{BagType}(a)^* \equiv a \in \text{Obj} \land \{\text{bound}, \text{bagdata}\} \subseteq \text{dom} a \land a \triangleq \text{bound} \in \mathbb{N} \land a \triangleq \text{bagdata} \in \text{bag}(\text{ELEMENTS})
\]

This can be seen by the following proof based on [Utt92, p49]:

\[
\begin{align*}
\text{BagType}(a)^* & \equiv \text{Monotonic Closure (3.2)} \\
& \equiv \exists a'. a' \in \text{Obj} \land \text{dom} a' = \{\text{bound}, \text{bagdata}\} \land \ a' \triangleq \text{bound} \in \mathbb{N} \land a' \triangleq \text{bagdata} \in \text{bag}(\text{ELEMENTS}) \land \ a \preceq a'
\end{align*}
\]

\(^2\text{When examining Utting and Robinson’s work, care is required as the syntax for their subtype relationship } \preceq \text{ is syntactically reversed when compared to the syntax for the subtyping relationship used in this thesis: } a \preceq b \equiv a \geq b.\)
\[ a \in \text{Obj} \land \text{dom } a = \{\text{bound}, \text{bagdata}\} \land \]
\[ a \preceq a' \]

- **Non-object fields of objects subtype discretely (reflexively).**

  \[ \exists a' : a' \in \text{Obj} \land \text{dom } a' = \{\text{bound}, \text{bagdata}\} \land \]
  \[ a \preceq a' \]

- **Expand Subtype definition**

  \[ \exists a' : a' \in \text{Obj} \land \text{dom } a' = \{\text{bound}, \text{bagdata}\} \land \]
  \[ a \in \text{Obj} \land a' \subseteq \text{dom } a \land \]
  \[ \forall i : \text{dom } a' \cdot a_{\circ}i \preceq a'_{\circ}i \]

- **Predicate Calculus**

  \[ \exists a' : a' \in \text{Obj} \land \text{dom } a' = \{\text{bound}, \text{bagdata}\} \land \]
  \[ a \in \text{Obj} \land \{\text{bound}, \text{bagdata}\} \subseteq \text{dom } a \land \]
  \[ \forall i : \text{dom } a' \cdot a_{\circ}i \preceq a'_{\circ}i \]

- **Existence of a superobject \(a'\) of \(a\).**

  \[ a \in \text{Obj} \land \{\text{bound}, \text{bagdata}\} \subseteq \text{dom } a \land \]
  \[ a_{\circ}\text{bound} \in \mathbb{N} \land a_{\circ}\text{bagdata} \in \text{bag}(\text{ELEMENTS}) \]

The predicate \(\text{BagMC}(a)\) is the monotonically closed analogy of the \(\text{Bag}(a)\) predicate:

\[ \text{BagMC}(a) \triangleq \text{BagType}(a) \land \#(a_{\circ}\text{bagdata}) \leq a_{\circ}\text{bound} \]

This predicate holds for all subobjects of those specified by \(\text{Bag}(a)\).

The *put* method is used to add an element *el* to the bag contained in the object *host*.

\[
\text{host}: \text{put}_{\text{Bag}}(\text{value } el : \text{ELEMENTS}) \triangleq
\begin{align*}
\text{BagMC}(\text{host}) \land el & \in \text{ELEMENTS} \land \\
\#(\text{host}_{\circ}\text{bagdata}) & < \text{host}_{\circ}\text{bound}
\end{align*}
\]

\[
\text{frame}: [\text{pre}, \text{post}] \equiv \text{frame}: \left[\begin{array}{c}
\text{pre} \\
\text{post}
\end{array}\right]
\]

There is a precondition on the operation that ensures the cardinality of the bag is less than the bound. Since the entire object *host* is in the frame, constraints are required for any fields that are to be kept invariant; hence the postcondition: \(\text{host}_{\circ}\text{bound} = \text{host}_{\circ}\text{bound}\).

The final conjunct of the postcondition states that the final value of the \(\text{bagdata}\) field of *host* is the original bag unioned with a singleton bag containing *el*, that is \([el]\). The symbol \(\uplus\) represents bag union. The postcondition of the specification does not constrain the additional fields of subobjects. It is unclear how to preclude the arbitrary modification of those fields.
The \( \text{getBag} \) method uses \( host \) again as the host object argument, yet uses \( el \) as a result parameter to return an element from the bag.

\[
\text{host.getBag}(\text{result } el : \text{ELEMENTS}) \triangleq \\
\begin{align*}
\text{BagMC}(\text{host}) \land \text{host} \circ \text{bagdata} \neq [] \\
\text{host} \circ \text{bagdata} = \text{host} \circ \text{bagdata} \cup [el]
\end{align*}
\]

The precondition \( \text{host} \circ \text{bagdata} \neq [] \) allows undefined behaviour if the bag is empty. The final conjunct of the postcondition nondeterministically chooses \( el \) to be an element of \( \text{bagdata} \) and simultaneously removes it.

The \( \text{cardBag} \) method uses \( host \) as the host object parameter and \( num \) as a result parameter for the cardinality of the bag.

\[
\text{host.cardBag}(\text{result } num : \mathbb{N}) \triangleq \\
\begin{align*}
\text{BagMC}(\text{host}) \\
\text{num} = \#(\text{host} \circ \text{bagdata})
\end{align*}
\]

Given \( Bag \) objects and the above three methods, the specification is completed by providing an appropriate late binding function \( \psi \) (method name \( \rightarrow \) host objects \( \rightarrow \) method). In this case the following function is sufficient.

\[
\psi \triangleq \begin{cases}
\text{put} \mapsto \{ \text{Bag} \mapsto \text{putBag} \}, \\
\text{get} \mapsto \{ \text{Bag} \mapsto \text{getBag} \}, \\
\text{card} \mapsto \{ \text{Bag} \mapsto \text{cardBag} \}
\end{cases}
\]

Consider a subtype \( \text{SubBag} \) with corresponding methods \( \text{putSubBag}, \text{getSubBag} \) and \( \text{cardSubBag} \). If \( \text{putBag} \sqsubseteq \text{putSubBag}, \text{getBag} \sqsubseteq \text{getSubBag} \) and \( \text{cardBag} \sqsubseteq \text{cardSubBag} \) then the following \( \psi \) function is monotonic (in refinement) with respect to subtyping.

\[
\psi \triangleq \begin{cases}
\text{put} \mapsto \{ \text{Bag} \mapsto \text{putBag}, \text{SubBag} \mapsto \text{putSubBag} \}, \\
\text{get} \mapsto \{ \text{Bag} \mapsto \text{getBag}, \text{SubBag} \mapsto \text{getSubBag} \}, \\
\text{card} \mapsto \{ \text{Bag} \mapsto \text{cardBag}, \text{SubBag} \mapsto \text{cardSubBag} \}
\end{cases}
\]

This function models single, dynamic dispatch. Multiple dispatch can be achieved by using multiple objects in the determination of which method to execute.

Using this model as a basis, the algorithmic refinement laws of the refinement calculus can be used for the development of programs.

\section{Sums of Products}

An alternative approach taken by Mikhajlova and Sekerinski [MS97] extends the typed lambda calculus with \textit{sum types} and \textit{product types} to produce a lattice-theoretic semantics.
Sum types, or disjoint unions, are used to combine multiple types while maintaining the distinctiveness of each type. The sum of two types $\sigma$ and $\tau$ is written as $\sigma + \tau$ while $\sigma$ and $\tau$ are termed the base types. The summation is undefined if the types are not disjoint.

Sum types are used by Mikhajlova and Sekerinski for modelling subtyping polymorphism and dynamic binding. They achieve polymorphism by using a statement summation operator defined by Back and Butler [BB95]. Building towards the statement summation operator, Back and Butler define a predicate summation operator. Given a predicate $p_1 : \text{Pred} \sigma$, where $\text{Pred} \sigma$ is a function mapping elements of $\sigma$ to the Booleans, and $p_2 : \text{Pred} \tau$, $p_1 + p_2$ acts as $p_1$ when given a state on $\sigma$ and as $p_2$ when given a state on $\tau$. In terms of this operator they define the statement summation operator. Consider a statement $S$ of type $\text{Tran} \sigma_{\text{pre}} \sigma_{\text{post}}$ where $\text{Tran} \sigma_{\text{pre}} \sigma_{\text{post}}$ is a function from a predicate of type $\text{Pred} \sigma_{\text{post}}$ to the weakest precondition predicate of type $\text{Pred} \sigma_{\text{pre}}$. Summing $S$ with $T : \text{Tran} \tau_{\text{pre}} \tau_{\text{post}}$, written $S + T : \text{Tran} \sigma_{\text{pre}} + \tau_{\text{pre}} \sigma_{\text{post}} + \tau_{\text{post}}$, is defined as:

$$(S + T) q \triangleq ((S; \langle \beta_{\sigma_{\text{post}}} \rangle) q) + ((T; \langle \beta_{\tau_{\text{post}}} \rangle) q)$$

for $q : \text{Pred} (\sigma_{\text{post}} + \tau_{\text{post}})$ where $\langle \beta_{\sigma_{\text{post}}} \rangle$ and $\langle \beta_{\tau_{\text{post}}} \rangle$ are used to inject states of type $\sigma_{\text{post}}$ and $\tau_{\text{post}}$ respectively into the type $\sigma_{\text{post}} + \tau_{\text{post}}$. When $S + T$ is executed in a state $\sigma_{\text{pre}}$ it acts as $S$. When executed in a state $\tau_{\text{pre}}$ it acts as $T$. To effect late binding of an object’s method, all versions of the method within a class hierarchy are summed together. The method invoked is determined statically by the type of the state space, which, in this context, is effectively the type of the object. The most important property of the summation operator is monotonicity:

$$S \subseteq S' \land T \subseteq T' \Rightarrow (S + T) \subseteq (S' + T')$$

This ensures that if a class’s method is refined, the late binding call which uses the method is also refined.

Mikhajlova and Sekerinski chose to represent their objects using product types. Product types are also known as Cartesian products or tuples.

**Example 3.3 (Objects as Products)** This example presents the bag example from the previous section using the object representation as provided by Mikhajlova and Sekerinski. To model the bag, two fields are used: a bound of type $\mathbb{N}$ and a bag data field of type $\text{bag}(\text{ELEMENTS})$. The object type is the Cartesian product of the fields:

$$\mathbb{N} \times \text{bag}(\text{ELEMENTS})$$

An element of this type is $(5, [])$. The field labels are not part of the object—this information is maintained in the ordering of the object. Mikhajlova and Sekerinski use the loss of labelling to encapsulate the fields of the object.

---

3The state types of the $\text{Tran} \sigma_{\text{pre}} \sigma_{\text{post}}$ are syntactically reversed when compared to the $\text{Ptrans} \sigma_{\text{post}} \sigma_{\text{pre}}$ syntax provided in Chapter 4.
A subtyping relation (and thus subsumption) is defined so that an element of a base type is also an element of a sum type that incorporates that base type. For example, given an object \( o_1 : O_1 \) and object type \( O_2 \) then via the subtyping relation and subsumption

\[
o_1 : (O_1 + O_2)
\]

Mikhajlova and Sekerinski use classes to couple an object’s type with the methods its elements are to invoke. Classes also incorporate a constructor method which is used to perform initialisation—typically by manipulation of an object’s fields. Class definition therefore involves specification of the fields, a constructor method, and the other class methods:

\[
C = \text{class} \quad \text{field } f_1 : F_1, \ldots, f_m : F_m \\
C(p : P) = S \\
\text{Meth}_1(i_1 : I_1) : O_1 = \text{Body}_1 \\
\vdots \\
\text{Meth}_n(i_n : I_n) : O_n = \text{Body}_n \\
\text{end}
\]

where \( f_1, \ldots, f_m \) are the fields and \( F_1, \ldots, F_m \) are their respective types, \( C \) is the name of the class and also the constructor method, \( p : P \) is used to pass parameters to the constructor, \( S \) is the body of the constructor method, \( i_1 : I_1, \ldots, i_n : I_n \) are the value parameters of the methods and \( res : O_1, \ldots, res : O_n \) are the result parameters. Assignment within \( \text{Body}_i \) to the variable \( res \) returns the value of the assignment as the result of the method.

**Definition 3.4 (Classes as Tuples)** Classes are represented as the tuple

\[
(S, \text{Body}_1, \ldots, \text{Body}_n)
\]

The type of the constructor \( S \) is

\[
S : \text{Tran}(F_1 \times \ldots \times F_m \times P)(F_1 \times \ldots \times F_m \times P)
\]

The constructor is modified with state enlargement and reduction commands [MS97] to alter its type to \( \text{Tran} P(F_1 \times \ldots \times F_m) \).

The types of the methods \( \text{Body}_{i \in 1..n} \) are:

\[
\text{Body}_i : \text{Tran}(F_1 \times \ldots \times F_m \times I_i \times O_i)(F_1 \times \ldots \times F_m \times I_i \times O_i)
\]

Similar to the constructor, each method is subjected to state transformation statements changing their type to

\[
\text{Tran}(F_1 \times \ldots \times F_m \times I_i)(F_1 \times \ldots \times F_m \times O_i)
\]
Since late binding is (partially) determined by the product of the object’s field types, it is not clear how Mikhajlova and Sekerinski can model the late binding of subclass instances in which no attributes are added, yet methods have been overwritten.

To ensure behavioural conformance of polymorphic objects, a constraint is placed on the declaration of a subclass. Given classes C and D, if class D is declared as a subclass of C then a proof obligation is generated requiring the developer to ensure that D is a class refinement of C. Class refinement is based on data refinement. Consequently, an abstraction relation is required to provide a link between the abstract and concrete fields.

**Definition 3.5 (Class Refinement of Sum Types)** Omitting state modification technicalities, class refinement is defined as follows. If \( C = (S, \text{Body}_1, ... \text{Body}_m) \)

\( D = (S', \text{Body}'_1, ..., \text{Body}'_m) \) and \( \tau(C) \) and \( \tau(D) \) are the types of \( C \) and \( D \) respectively then:

\[
C \sqsubseteq D \triangleq \exists R : \tau(D) \leftrightarrow \tau(C) \bullet \{ \pi_p \}; \ S \sqsubseteq S'; \ \{ R \} \land \\
\forall j : 1..m \bullet \{ R \times \pi_{i_j} \}; \ \text{Body}_j \sqsubseteq \text{Body}'_j; \ \{ R \times \iota_{O_j} \}
\]

where \( R \) is an abstraction relation used to map the fields of \( D \) to those of \( C \). \( \pi_p \) is the projection that maps the constructor parameters, \( \pi_{i_j} \) maps the input parameters, \( \iota_{O_j} \) maps the output parameters and \( \{ Q \} \) is defined such that it forms a predicate transformer from the relation \( Q \).

Mikhajlova and Sekerinski also provide a relationship between classes that potentially have entirely different method parameter types. They term the relationship interface refinement.

### 3.3 Storing Procedures in Variables

Using a set-theoretic model, Naumann [Nau94a, Nau94b] defines a higher-order language that supports extensible records and procedure-type variables. Procedure-type variables allow procedures to be stored in variables, and be passed as parameters to other procedures. Late binding can be modelled using these language features.

Placing methods inside objects which manipulate themselves complicates the typing of the objects. Research has provided typings for ‘objects’ which manipulate themselves yet do not contain specifications [Sch88] and for ‘objects’ that contain specifications yet do not do manipulate themselves [GHP95] but not both. Naumann simplifies the typing of the objects by imposing restrictions. Procedures cannot be recursive and the only external variables that may be modified are those that are global.

The definition of a call on a procedure variable and consequently an object’s method is straightforward.
Definition 3.6 (Procedure Variable Call) Given predicate \( post \), procedure variable \( pv \), and the denotation brackets \( [[ \cdot ]] \), then the predicate transformer \( [[\text{call} \ pv]] \) is defined as follows.

\[
[[\text{call} \ pv]]. post \triangleq pv.post
\]

Naumann introduces a subtyping relation\(^4\) that is reflexive, allows records to be extended and field types to subtype covariantly. In contrast with the approaches presented earlier, Naumann’s work does not provide a generalised subsumption rule. That is, the following rule of subsumption is not a rule in Naumann’s logic. The following rules are simplified versions provided for readability. Given object \( a \) and object types \( A \) and \( B \):

\[
\begin{align*}
& a : A \quad A \preceq B \\
& \quad a : B \\
\end{align*}
\]

The absence of this rule is deemed by Naumann to simplify the typing system. The role of subsumption is instead carried by several typing rules. The assignment rule allows objects to be assigned instances of subtypes. Given object types \( A \) and \( B \) where \( A \preceq B \), variable \( b \) of type \( B \) and object \( a : A \) then \( b \) can be assigned the instance \( a \) of subtype \( A \):

\[
\begin{align*}
& b : B \quad a : A \quad A \preceq B \\
& \quad b := a : \text{com} \quad \text{(assign)}
\end{align*}
\]

where \( \text{com} \) is the type of well-formed statements or commands. There are two similar rules which support this typing rule in its role of subsumption. The call rule allows value parameters to vary covariantly and the type guard rule provides a type for a type case statement.

The work of Naumann and Cavalcanti [CN00] uses a similar set-theoretic semantics. They define object states as tuples of attributes, methods similar to a procedure with a self parameter, and classes as tuples of methods. Consequently, methods are decoupled from fields. References and mutually recursive classes are not supported. They do, however, support type casts and type checks as predicates but they do not support type cast statements or type check statements. Even the support for type cast and type check predicates imposes constraints on their class refinement relation. For related technical reasons, they define class refinement in the following (paraphased) manner [CN00, p26]. A class \( C \) is refined by a class \( D \) if for any statement, \( s \), all statements \( s' \) which can be obtained by replacing some occurrences of \( C \) with \( D \) must be refinements of \( s \), i.e., \( s \sqsubseteq s' \). The practical use of this definition is not explored.

### 3.4 Miscellaneous Approaches

This section presents a summary of the work of other researchers investigating object-oriented verification and refinement logics.

\(^4\)Naumann originally used the syntax: \( \in \). This thesis uses the \( \preceq \) syntax for consistency.
The work of Back, Mikhajlova and von Wright [BMvW00] is similar to that of Mikhajlova and Sekerinski’s [MS97]. They provide support for dynamic binding and non-recursive classes. Although different to that of Mikhajlova and Sekerinski [MS97], their definition of class refinement is also similar to data refinement. Clients are modelled as an iterative choice of method invocations. Using this model they show that class refinement implies refinement of clients. This property is generalised by Theorem 7.27 on page 91 of this thesis. The generalisation is termed construction monotonicity.

Earlier work by Sekerinski [Sek96] investigated the use of packed record types in the formation of an object-oriented refinement calculus framework. Packed record types consist of two records, one for methods and one for fields. The records are then ‘packed’ so that the fields are hidden and the methods can use field records that are possibly extended. The approach was abandoned as the invocation of methods involved complex unpacking and repacking of objects.

Lewerentz et al. [LLRS95] use the classical lattice-theoretic refinement calculus as the foundation for a logic in which lattices of classes can be formed. Then, given two classes, they show that a superclass and subclass of these classes can be calculated by using the lattice concepts of meet and join.

Leino [Lei95] used Dijkstra’s Guarded Command Language as the foundation for an object-oriented verificational logic. Leino [Lei97] subsequently used a weakest liberal precondition semantics to provide an axiomatic semantics for an object-oriented programming language called Ecstatic. A weakest liberal precondition semantics is similar to a weakest precondition semantics except that termination is not considered.

Abadi and Leino [AL97] use Hoare logic to provide a wide-spectrum, object-based programming language and verificational logic.

\begin{verbatim}
x false | true if x then a_0 else a_1 let x = a in b [f_i = x^i_{1..n}, m_j = \zeta(y_j) b^j_{1..m}] x.f x.m x.f := y
\end{verbatim}

The language is typed and permits subtyping, subsumption and inheritance. An operational semantics is provided and a soundness theorem is given that guarantees an outcome of a program derived using the axiomatic semantics is the outcome achieved using the operational semantics.

The language above is extended with specifications that are mutations of Hoare triples. The Hoare triple

\begin{verbatim}
{pre} s {post}
\end{verbatim}

5Using existential quantification of types as discussed by Abadi and Cardelli [AC96].
is represented using the transition relation

\[ s : S :: \text{pre} \Rightarrow \text{post} \]

where \( S \) is the type of \( s \). Within Hoare triples, \( s \) is a statement. In transition relations, \( s \) may also be a constant, variable or expression. Using transition relations, a Hoare-like logic is developed for the language. Like Hoare logic, the resulting axiomatic semantics guarantees only partial correctness. Verification can be performed using the axiomatic semantics. Unfortunately, the axiomatic semantics is not complete; hence many verifications cannot be performed despite the correctness of the program.

While Abadi and Leino mutate Hoare triples into transition relations, de Figueiredo [dF95] extends Hoare triples by including a result condition in the traditional triple. The specification

\[ \{p\}c\{q,t\} \]

denotes \( p \) as the precondition, \( c \) as the programming language command, \( q \) as the post-condition and \( t \) as the condition whose interpretation denotes the value returned. The inclusion of the \( t \) result is motivated by the fact that the commands of object-oriented languages can modify a state (hence \( p \) and \( q \)) and also return a value (hence \( t \)). The Hoare triple extension provides the foundation for an object-oriented verificational calculus.

Ahmed and Morris [AM91] use Martin-Löf’s type theory to define a module refinement calculus. They introduce propositional expressions into types to allow a module’s type to be its specification. This “allows us to specify the detailed behaviour of objects purely by their type.” Unfortunately, the authors admit the refinement definition is not adequate and can not refine away some intermediary constructs introduced during refinement. This means that many intuitive refinements can not be proven.

### 3.5 Conclusions

The theoretical foundations of existing object-oriented refinement calculi vary significantly. Additionally, the pervasive effects of object representation means that the resulting object-oriented refinement calculi semantics are so divergent that it is difficult to compare the calculi except on a feature-by-feature comparison. The divergent nature of the foundations also makes it difficult to translate properties and techniques from one approach to another. Even the syntax of related concepts varies significantly. This problem is glossed over in this chapter as the syntax has been made more consistent.

The majority of the approaches summarised here decouple fields from methods to simplify the typings of objects. Chapter 5 takes the alternative approach and incorporates methods into objects. This approach is intended to simplify the semantics for late binding and to capture the inherent encapsulation principles of object orientation.
The complications that decoupling fields from methods introduces are indicated by the need to ‘compile’ the late binding mechanism. For Utting’s [Utt92] function, the introduction of a new class requires the ‘recompilation’ of the function. The loss of encapsulation in Mikhajlova and Sekerinski’s [MS97] sum types approach is illustrated by the fact that classes must be aware of all their subclasses. The semantics of a class’s method is determined by summing all corresponding methods from all classes in the class hierarchy. Consequently adding a new subclass requires ‘recompilation’ of all superclasses. While there are no known practical side effects, it is disconcerting that a class must be aware of its subclasses, that its semantics are altered by the introduction of a new class, and that ‘recompilation’ of superclasses of the new class is, at least in theory, required.

The slight complication of Utting’s late binding mechanism, is outweighed however, when its flexibility is considered. Not only does it handle multiple inheritance but it can also be considered a generalisation of the late binding mechanism that is employed by all approaches that decouple fields from methods. For example, given that the sum type approach is analogous to dynamic type checking [BB95, p5] it is a special case of Utting’s function which type checks the host.

Just as techniques that decouple methods have complications, so do techniques that embed methods into objects. Chapter 5 details the constraints that are currently imposed given the object representation presented there. These constraints are similar to those imposed on the earlier (embedded methods) work of Naumann [Nau94b]. In comparison with the object-oriented refinement calculus presented in Chapter 7, Naumann does not abstract away from the complexities of the type system. Consequently, the intricacies of the type system pervades the remainder of his calculus. An advantage of Naumann’s earlier work is that it supports procedure variables: a concept, itself worthy of study.

The complexities of the presented object-oriented refinement calculi are caused by the manner in which researchers intertwine inherently complex objects and the refinement calculus. The remainder of this thesis takes an alternative approach: abstraction. Rather than forcing the refinement calculus to suffer because of the theoretical foundations of objects, it seems sensible to provide an abstraction of objects. By identifying desired properties of objects and building the object-oriented refinement calculus on this foundation the resulting calculus should be both simpler and the properties and techniques more reusable.

The identification of an appropriate object representation is consequently an orthogonal task to the development of an appropriate object-oriented refinement calculus. Given the exhaustive research of Abadi and Cardelli [AC96], their calculi (in particular $\text{FOb}_{\leq \mu}$) is a reasonable starting point for theidentification of the desirable properties of objects. To demonstrate that the object-oriented refinement calculus developed by the thesis can be grounded, the thesis presents a specific object representation (Chapter 5).
Chapter 4

A Typed Refinement Calculus

This chapter introduces a typed refinement calculus that supports a general subtyping relation. This is achieved by adding type information to the classical refinement calculus definitions. In subsequent chapters, predicate transformers are embedded into objects. To determine the type of such an object (and hence to support subtyping) the types of the embedded predicate transformers must be known. The chapter culminates in the definition of a refinement relation that relates heterogeneously typed predicate transformers, i.e., predicate transformers with different types. To the author’s knowledge, this is the first treatment of such a relation. Given that the modifications of the definitions only add type information, the results of classical refinement calculi are maintained.

The purpose of this chapter is to provide a foundation upon which the object-oriented language of Chapter 6 can be built. Like the object-oriented refinement calculi of Utting and Robinson [UR92, Utt92], Mikhajlova and Sekerinski [MS97], and Cavalcanti and Naumann [CN00], the framework of this object-oriented language is based on a lattice-theoretic foundation. The reader is referred to Appendix C for an introduction to lattice theory. The lattice-theoretic model is developed from the Booleans. The Booleans, \{false, true\}, form a complete Boolean lattice under the ordering false \(\Rightarrow\) true. Many of the lattice properties are extended to predicates and predicate transformers by pointwise extension as described by Back and von Wright [BvW98, p128,p190].

This foundation is complemented with the object types introduced in Section 4.1. To allow for the composition of objects, Section 4.2 introduces the concepts of meet and join for object types. Section 4.3 provides the type models for states, predicates and predicate transformers. These models are based on classical techniques modified to allow for both typing and subtyping. While typing and subtyping rules have been provided before, to the author’s knowledge, this presentation is the first to explore the definitions of heterogeneously typed connectives. Section 4.3 also introduces a novel definition that permits refinement of statements of heterogeneous types.
4.1 Object Theory and Calculi

The $\lambda$-calculus [Bar81] is a theory of functions which can be used to provide the foundations for procedural languages. Object calculi, analogously, are used to provide the foundations for object-oriented languages. Object calculi reduce object-oriented notions such as subtyping and inheritance to a few basic concepts. Abadi and Cardelli [AC96] provide a collection of such object-oriented calculi. Their book advances through the calculi by presenting one calculus, showing its usefulness, and using its deficiencies to motivate the next. The book also explores the topic of a denotational semantics for the object calculi, and various techniques for defining and axiomatising the ‘Self’ type of an object. Of interest for this research is their discussion of interpreting objects. They address the question of “Why should we study object calculi, instead of encoding objects in $\lambda$-calculi” [AC96, p77,257]. The aim of this thesis is to utilise their advances in object calculi to suggest improvements in the modelling of object-oriented refinement calculi.

From the assortment of calculi presented by Abadi and Cardelli [AC96], a calculus had to be chosen that would most easily integrate with the lattice-theoretic foundation of a refinement calculus. The chosen calculus is $\text{FOb}_{\leq,\mu}$. The rationale for this choice is discussed in Chapter 5 after the required concepts have been introduced. This section presents an introduction to the object calculus features that are used for the integration of the $\text{FOb}_{\leq,\mu}$ calculus with the refinement calculus, namely object types, function types and record types.

The form of the rules and judgements used by the object calculus is as follows. Each rule has a number of premise judgements above the line and a consequence judgement below the line.

\[
\begin{array}{c}
\text{premise}_1 \\
\text{premise}_2 \\
\hline \\
\text{consequence}_1
\end{array}
\]

**Judgements** Each judgement $E \vdash A$ consists of an environment $E$ for the assertion $A$. A judgement of the form $E \vdash \diamond$ expresses that $E$ is a well-formed environment. Given a type construction $T$, the judgement $E \vdash T$ expresses that $T$ is a well-formed type in environment $E$. Other forms of judgements used include:

**Subtyping:** $E \vdash A \leq B$ states that $A$ is a subtype of $B$ in environment $E$,

**Value typing:** $E \vdash a : A$ states that $a$ is of type $A$ in environment $E$, and finally,

**Truth:** $E \vdash p$ states that $p$ is true in the environment $E$, e.g., $\{a : A\} \cup \{A \leq B\} \vdash a : B$.

**Syntactic Definitions** Syntactic definitions are provided using the syntax $\equiv$. They are occasionally presented in rule form. This notation assumes the premises denote the
well-formedness constraints of the construct. E.g., the syntax

\[
\frac{\vdash p}{A \equiv B}
\]

denotes that \(A\) is syntactically defined as \(B\) and is well-formed when the premise \(\vdash p\) holds.

**Environment Omission**  When the environment of all premises and the consequence is the same and is evident from the context (or is irrelevant), the environment and the judgement syntax can be omitted. For example, the syntax

\[
\frac{A_1}{A_2}
\]

may be used in lieu of the following.

\[
\frac{E \vdash A_1}{E \vdash A_2}
\]

**Rule Abbreviations**  When writing rules, the abbreviation for all \((i \in 1..n) \bullet E_i \vdash A_i\) (where \(n > 0\)) stands for \(n\) premises of the form \(E_1 \vdash A_1 \ldots E_n \vdash A_n\). A premise of the form \(j \in 1..n\), in comparison, denotes that there are \(n\) different rules, one for each \(1..n\). For example, the rule

\[
\frac{A_j, j \in 1..n}{B_j}
\]

is actually the following set of rules.

\[
\frac{A_1 \ldots A_j \ldots A_n}{B_1 \ldots B_j \ldots B_n}
\]

**Top**  The weakest type is \(\text{Top}\). It contains no attributes and it is a supertype of all other types.

**Environments**  Environments are formed by the combination of value-type environments and type environments.

\[
\text{Environment} \equiv \text{ValueTypes} \times \text{Subtypes}
\]

A value-type environment is a relation from labels to types representing the declared type of each label (variable).

\[
\text{ValueTypes} \equiv \text{Labels} \rightarrow \text{Types}
\]
Although the type of variables may not be necessarily unique (due to subtyping), the environment stores the declared type. The syntax \( a : A \) is used in the environment to denote a mapping of the variable \( a \) to the type \( A \), i.e., \( a \) has type \( A \).

A type environment is a relation from type labels to types representing subtype relationships.

\[
\text{Subtypes} \equiv \text{Labels} \leftrightarrow \text{Types}
\]

The syntax \( A \preceq B \) is used in the environment to denote a mapping from \( A \) to the type \( B \), i.e., \( A \) is a subtype of \( B \).

**Free Variables** \( T \notin A \) denotes that \( T \) does not occur free in \( A \).

**Substitution** Syntactic substitutions are effected using the notation \( A[x/y] \) which denotes the substitution of \( y \) for \( x \) in \( A \).

The object calculus introduces constructs termed objects. Objects differ from records (or functions) in that each component attribute has access to its own host. Objects are typed and a subtyping relationship is introduced. The subtyping relationship \( (\preceq) \) is defined as set inclusion within the object calculus semantics.

The syntax of the object calculus types is:

\[
\text{TYPE ::= X} \quad \text{– type identifier}
\]
\[
\text{Top} \quad \text{– the greatest type}
\]
\[
\text{TYPE \to TYPE} \quad \text{– function type}
\]
\[
\mu(X)\text{TYPE} \quad \text{– recursive type}
\]
\[
\forall(X \preceq \text{TYPE})\text{TYPE} \quad \text{– bounded universal type}
\]
\[
\exists(X \preceq \text{TYPE})\text{TYPE} \quad \text{– bounded existential type}
\]
\[
\text{Obj \{ Ident : TYPE, \ldots \}} \quad \text{– object type}
\]
\[
\text{\{Ident : TYPE, \ldots\}_{rt}} \quad \text{– record type}
\]

Function types, object types, and record types are introduced in this chapter. Universal and existential types are not directly used by this thesis. Their syntax is presented here as they are used in Appendix A for the interpretation of variantly annotated object types. Variantly annotated object types are introduced in Chapter 5, as are recursive types.

The syntax of the object calculus terms is:
CHAPTER 4. A TYPED REFINEMENT CALCULUS

\[
\begin{align*}
\text{TERM} & ::= \ x & \quad & \text{– identifier} \\
\lambda(x : \text{TYPE})\text{TERM} & \quad & \text{– function abstraction} \\
\text{TERM}(\text{TERM}) & \quad & \text{– function application} \\
\text{fold}(&\text{TYPE}, \text{TERM}) & \quad & \text{– recursive fold} \\
\text{unfold}(\text{TERM}) & \quad & \text{– recursive unfold} \\
\lambda(X \triangleq \text{TYPE})\text{TERM} & \quad & \text{– type abstraction} \\
\text{TERM}(\text{TYPE}) & \quad & \text{– type application} \\
p\text{ack} \ X \triangleq \text{TYPE} = \text{TYPE with} & \quad \text{– packaging} \\
\text{TERM : TYPE} & \quad & \\
\text{open} \ \text{TERM as} X \triangleq \text{TYPE}, & \quad \text{– opening} \\
x : \text{TYPE in} \ \text{TERM : TYPE} & \quad & \\
\text{object} \ \{ \ \text{Ident} = \text{METHOD}, \ldots \} & \quad \text{– object construction} \\
\text{TERM} \in \text{Ident} & \quad \text{– object method invocations} \\
\text{TERM} \triangleq \text{METHOD} & \quad \text{– object method update} \\
\{ \text{Ident} = \text{TERM} ; \ldots \} & \quad \text{– record construction} \\
\text{TERM} \ \text{Ident} & \quad \text{– record selection} \\
\text{TERM} \ \text{Ident} : = \ \text{TERM} & \quad \text{– functional record update} \\
\end{align*}
\]

\[
\text{METHOD} ::= \ \zeta(x : \text{TYPE}) \ \text{TERM}
\]

Object Types  Object types have the syntax \( \text{Obj} \ F \) where \( F \) is a function from the labels specifying the names of the attributes to their respective types: Labels \( \rightarrow \) Types. For the rules in this section and those of Appendix A, both \( F \) and \( G \) are functions of the type \( \text{Label} \rightarrow \text{Types} \). For example, objects of type \( \text{Obj} \ \{ \ \text{Ident} : \mathbb{N} \} \) have an attribute \( l \) of type \( \mathbb{N} \).

The object calculus contains many rules for manipulating object calculus constructs and for determining various properties of those constructs. For example, the rule which can be used to show that such types are well-formed (syntactically and semantically ‘valid’) is termed Type Object. All properties that present well-formed types have names with the prefix ‘Type’. Other rule name prefixes are

- ‘Eq’ for properties that equate values—equality properties;
- ‘Eval’ for evaluation—properties that reduce or evaluate values that are not in their simplest form;
- ‘Val’ for properties that present the types of values;
- ‘Sub’ for subtyping properties; and
- ‘Env’ for rules that allow well-formed environments to be constructed.

The following rule is used to determine whether an object type is well-formed.
Axiom 4.1 (Type Object) The object type $Obj \{ \; i \in 1..n \cdot \beta_i \; \}$ is well-formed if the type of each of its attributes is well-formed.

\[
\text{for all } (i \in 1..n) \bullet \beta_i \\
Obj \{ \; i \in 1..n \cdot \beta_i \; \}
\]

The derivation of such rules is outside the scope of this thesis. Interested readers are referred to the work of Abadi and Cardelli [AC96].

Methods Methods have the syntax $\varsigma(x : X) \text{ body}$ where $\varsigma$ is an object quantifier analogous to the $\lambda$ quantifier in the lambda calculus. The term $x$ is the self parameter and $X$ is its type. Whenever a method is invoked, $x$ is bound to the host object. The body of the method, $\text{body}$, may reference $x$. Consequently the behaviour of a method is dependent upon the context, or host, in which the method is invoked. This, in essence, simulates the behaviour of dynamic dispatch. Methods that do not reference their self parameters, $x$, are termed fields.

Objects Objects are collections of labels and associated methods. The syntax of objects is

\[
\text{object } f
\]

where $f$ is a function mapping labels to methods. For example,

\[
\text{object } \{ \; i \in 1..n \cdot \beta_i = \varsigma(x : X) \text{ body}_i \; \}
\]

is a syntactically valid object.

All objects are typed. The following rule, Val Object, determines the type of an object using the types of the method (or field) bodies. The method types are collected and associated with the appropriate labels within the resulting object type. An alternative view of the following rule is that to show that an object is of a certain type, namely $Obj F$, one must show that the types of its attributes correspond to the relevant types in the object type.

Axiom 4.2 (Val Object) Given $\gamma \models Obj F$

\[
\text{for all } (l \in \text{dom}(F)) \bullet E \cup \{x : \gamma\} \vdash \text{body}_l : F(l) \\
E \vdash \text{object } \{ \; l \in \text{dom}(F) \cdot \beta_i = \varsigma(x : \gamma) \text{ body}_i \; \} : \gamma
\]

For example, to show that $\text{object } \{ \; 5 = \varsigma(x : X) \; \}$ is of type $Obj \{ \; l : \mathbb{N} \; \}$, one must show that $5 : \mathbb{N}$.
Subtyping  There is a characteristic property of subtyping (\(\preceq\)) called subsumption. This property is formalised by the rule **Val Subsumption**.

**Axiom 4.3 (Val Subsumption)** Given an object of a particular type \(\delta\), that object also has the type of any supertype of \(\delta\).

\[
\frac{d : \delta}{d : \gamma}
\]

An object does not change when it is is subsumed, rather the static information known about the object is reduced. Consequently, subsumption does not alter the behaviour of an object.

Subtyping is a partial order (reflexive, transitive and anti-symmetric). Basic types, such as the Booleans and the natural numbers, subtype only reflexively and as subtypes of \(\text{Top}\). Subtyping of object types occurs when the subtype has more attributes.

**Axiom 4.4 (Sub Object)**

\[
\frac{F \subseteq G}{\text{Obj } G \preceq \text{Obj } F}
\]

The second premise ensures the subtype is a well-formed object type.

**Most Defined Type**  Although not defined by the object calculus, a ‘most defined type’ judgement, \(a :_\bot B\), is introduced here to aid readability.

\[
a :_\bot B \overset{\triangleq}{=} a : B \land (\forall T \bullet a : T \Rightarrow B \preceq T)
\]

The environment stores the most defined static type of a variable, not its dynamic type. Hence rules that use the knowledge of a variable’s most defined type are actually constraining the environment. For an environment \(E\):

\[
\frac{\{a : B\} \cup E \vdash }{\{a : B\} \cup E \vdash a :_\bot B}
\]

The type judgement of an expression obtained without use of Val Subsumption (4.3) is the most defined type.

**Equivalence**  Equivalence relations typically relate values of the same type. However, due to subtyping and subsumption, objects can take on multiple types. Consequently, there are many equivalence relations, one for each type, or alternatively, the equivalence relation is parameterised by a type. The syntax \(c =_\beta d\) denotes that object \(c\) is equivalent to \(d\) under the type \(\beta\). A result of the parameterisation of equality is that while two objects may be equivalent under one type, they may not be equal under another.

For object types, two instances are equivalent if, when truncated to the same attributes (those of the type that parameterises the equivalence (\(\gamma\))), the respective attributes of both objects are equivalent.
Axiom 4.5 (Eq Object) Given $\gamma \equiv \text{Obj} \{ \ i \in 1..n \bullet l_i : \beta_i \ \}$, $p \geq 0$, and $q \geq 0$

for all $(i \in 1..n) \bullet E \cup \{ x : \gamma \} \vdash b_i = \beta_i b_i'$

$E \vdash \text{object} \{ \ i \in 1..n + p \bullet l_i = \varsigma(x : \gamma) b_i \} = \gamma \ \text{object} \{ \ i \in 1..n + q \bullet l_i = \varsigma(x : \gamma) b_i' \}$

The premise ensures the equivalence of the attributes given that the self parameter $(x)$ has type $\gamma$.

Method Invocation/Field Selection In this object calculus, fields and methods are treated identically. Thus method invocation is also field selection. Method selection is equivalent to ‘extracting’ the relevant method body and binding the self parameter. So for the object $\text{object} \{ \ l_1 = \varsigma(x : \gamma) 5, l_2 = \varsigma(y : \gamma) y \}$, selecting field $l_1$ first binds $x$ to the host object; however since there is no occurrence of $x$ in $5$ this binding has no effect. Consequently selecting $l_1$ returns the object $5$. In comparison, when method $l_2$ is invoked, $y$ is bound to the host object and hence the method returns $y$ which is the host object itself.

The name of the property for method invocation or field selection is Eval Select.

Axiom 4.6 (Eval Select) Given $\gamma \equiv \text{Obj} \{ \ i \in 1..n \bullet l_i : \beta_i \ \}$ and

$\ c \equiv \text{object} \{ \ i \in 1..n \bullet l_i = \varsigma(x : \gamma) b_i \ \}$

then

$\ c : \gamma \ j \in 1..n \ \
\ c \circ \ l_j = \beta_j b_j[x \backslash c] \ \\$

where $b_j[x \backslash c]$ stands for the substitution of the self parameter $x$ with $c$ in the method body $b_j$. The premise $j \in 1..n$ parameterises the rule thereby acting as though $n$ rules are introduced: one for each $j \in 1..n$. To help distinguish the use of the object calculus, the standard field selection and method invocation syntax (i.e., '.') is replaced with '$\circ$'.

For method invocation (and field selection), another relevant rule is Val Select. The type of a method invocation (field selection) is the type of the method body associated with the label being selected. For example, for the object type $\text{Obj} \{ \ l : \mathbb{N} \ \}$, the type of the method invocation of $l$ is $\mathbb{N}$.

Axiom 4.7 (Val Select)

$\ c : \text{Obj} F \quad l \in \text{dom}(F) \ \
\ c \circ \ l : F(l) \ \
$

The final property relevant to method invocations is Eq Select. It is desirable that given two equivalent objects, selecting the same label on each produces two objects that are also equivalent.

Axiom 4.8 (Eq Select) Given $\gamma \equiv \text{Obj} F$

$\ c =_\gamma c' \quad l \in \text{dom}(F) \ \
\ c \circ l =_{F(l)} c' \circ l \ \
$
Method Update The evaluation of a method update (field update) is intuitive: updating an object \( c \) at label \( l \) with body \( \varsigma(x: \gamma) b \), denoted by \( c \circ l \models \varsigma(x: \gamma) b \), returns an object which equals \( c \) at all labels, except at \( l \) which is now \( \varsigma(x: \gamma) b \). The rule **Eval Update** is used to evaluate the result of a method update\(^1\).

**Axiom 4.9 (Eval Update)** Given \( \gamma \models Obj F \) and \( c \models object f \) then

\[
E \vdash c : \gamma \quad E \cup \{x : \gamma\} \vdash b : F(l) \quad l \in \text{dom}(F)
\]

\[
E \vdash (c \circ l \models \varsigma(x: \gamma) b) =_\gamma \text{object } (f \oplus \{l = \varsigma(x: \gamma) b\})
\]

Function Types The object calculus has rules for determining the well-formedness of function types (A.4), for determining the type of a function (A.5), and for determining the type of a function application (A.6).

**Axiom 4.10 (Sub Arrow)** If \( B \) is a subtype of \( B' \) then the function type \( A \to B \) which results in an object of type \( B \) can also be considered to produce a type \( B' \). This is referred to as **covariance**. Since the function \( A \to B \) accepts objects of type \( A \), any subtype of \( A \) can be passed to the function. This is referred to as **contravariance**.

\[
A' \leq A \quad B \leq B' \\
\quad A \to B \leq A' \to B'
\]

There also exist equational rules for determining if two functions are equivalent (A.28), for determining if two function applications are equivalent (A.29), for beta-conversion (A.30) and for eta-conversion (A.31).

Record Types An object type \( O \) is a subtype of an object type \( P \) if \( O \) has the same attributes (of the same type) of \( P \) and possibly more. In contrast, the types of a record type’s attributes may vary covariantly.

**Axiom 4.11 (Sub Record)** Given an environment where \( rfun_{rt} \) and \( rfun'_{rt} \) are well-formed record types,

\[
E \vdash \text{dom}(rfun') \subseteq \text{dom}(rfun) \\
\quad \text{for all } (i \in \text{dom}(rfun')) \quad E \vdash rfun(i) \models rfun'(i) \\
\quad \text{for all } (i \in \text{dom}(rfun) \setminus \text{dom}(rfun')) \quad E \vdash rfun(i)
\]

\[
E \vdash rfun_{rt} \leq rfun'_{rt}
\]

\(^1\)Typically the rule is generalised to allow reasoning about objects with a different number of attributes. Here, for readability, the rule is simplified, allowing reasoning only about objects with the same number of attributes.
There also exist various typing and equational rules of record types of a similar nature to those for function types. A complete axiomisation is presented in Appendix A.3.

These rules form the core of the object calculus upon which our work is based. Using similar calculi, Abadi and Cardelli have succeeded in creating several object-oriented languages by providing translations from the language constructs into the object calculus constructs. These languages have varying levels of functionality; some are almost fully featured object-oriented languages.

4.2 Least-Upper and Greatest-Lower Bounds

The definitions for the greatest lower bound and least upper bound of collections of types are not part of the object calculus $\text{FOb}_{\equiv_{\mu}}$ developed by Abadi and Cardelli. They are developed here for determining the type of predicate connectives and predicate transformer connectives (as given in Section 4.3.1).

Definition 4.12 (Greatest Lower Bound) The greatest lower bound of types $\alpha$ and $\beta$, $(\alpha \sqcap \beta)$, is the greatest type that is a subtype of both.

\[
\begin{align*}
\alpha &\sqcap \beta \\
(\alpha \sqcap \beta) &\leq \alpha \\
(\alpha \sqcap \beta) &\leq \beta \\
\forall \gamma \bullet (\gamma \leq \alpha \land \gamma \leq \beta) &\Rightarrow \gamma \leq (\alpha \sqcap \beta)
\end{align*}
\]

The consequent $\forall \gamma \bullet (\gamma \leq \alpha \land \gamma \leq \beta) \Rightarrow \gamma \leq (\alpha \sqcap \beta)$ denotes that for all lower bounds ($\gamma$) of $\alpha$ and $\beta$, the greatest lower bound is greater (or equal): $\gamma \leq (\alpha \sqcap \beta)$. The premises ensure that the types $\alpha$, $\beta$ and $(\alpha \sqcap \beta)$ are well-formed. The well-formedness of $(\alpha \sqcap \beta)$ is equivalent to the existence of a type which is a subtype of both $\alpha$ and $\beta$:

\[
\exists \gamma \bullet \gamma \leq \alpha \land \gamma \leq \beta \\
(\alpha \sqcap \beta)
\]

The syntactic calculation of the greatest lower bound of two object types is obtained by combining the identifiers of both. For disjoint labels, the greatest lower bound can be determined by:

\[
(\text{Obj } F \sqcap \text{Obj } G) \cong \text{Obj } (F \cup G)
\]

When $\alpha$ and $\beta$ are not disjoint, the greatest lower bound of those attributes that are shared must be calculated. This is determined by the pointwise calculation of the greatest lower bounds of the corresponding attributes, provided that each of these is well-formed.

\[
\text{for all } (l \in \text{dom}(F) \cap \text{dom}(G)) \bullet (F(l) \sqcap G(l))
\]

The label-type mappings for labels only in $F$ are obtained using $(\text{dom}(G) \leq F)$ where $\leq$ is the domain subtraction operator. Similarly the label-type mappings for labels only in
G are obtained using \((\text{dom}(F) \triangleleft G)\). Collecting these together forms the greatest lower bound:

\[
\forall l \in (\text{dom}(F) \cap \text{dom}(G)) \bullet (F(l) \cap G(l))
\]

\[
(F \cap G) \equiv \{ l \in (\text{dom}(F) \cap \text{dom}(G)) \bullet l : (F(l) \cap G(l)) \} \cup \{ l \in (\text{dom}(G) \setminus F) \cup (\text{dom}(G) \setminus F) \}
\]

The premise ensures that the greatest lower bound for each pair of associated types is well-formed.

The greatest lower bound of different basic types is undefined. For example \((\mathbb{B} \cap \mathbb{N})\) is undefined. That is, there is no type which is a subtype of both the booleans and the natural numbers. The greatest lower bound of the same basic type is that same basic type. For example \((\mathbb{N} \cap \mathbb{N})\) is \(\mathbb{N}\).

It may appear counterintuitive that the syntax ‘\(\cap\)’ is used for unioning identifiers in object types. However, the following example illustrates that it is semantically intuitive even if syntactically it is not.

**Example 4.13 (Meet Syntax)** The object type \(\text{Obj} \{ a : \mathbb{B} \} \) contains all objects that have a field \(a\) of type \(\mathbb{B}\). Similarly, the object type \(\text{Obj} \{ b : \mathbb{B} \} \) contains all objects that possess a field \(b\) of type \(\mathbb{B}\). Using the greatest lower bound, a subtype of both can be calculated: \(\text{Obj} \{ a : \mathbb{B}, b : \mathbb{B} \} \). This object type contains all objects that possess a field \(a\) and a field \(b\), both of which are of type \(\mathbb{B}\). Therefore, all objects in the type \(\text{Obj} \{ a : \mathbb{B}, b : \mathbb{B} \} \) are in \(\text{Obj} \{ a : \mathbb{B} \} \cap \text{Obj} \{ b : \mathbb{B} \} \). The calculation of the greatest lower bound of two object types is actually an intersection of the sets of objects that form the types, even though syntactically the fields are unioned.

\[
\text{Obj} \{ a : \mathbb{B}, b : \mathbb{B} \} \equiv \text{Obj} \{ a : \mathbb{B} \} \cap \text{Obj} \{ b : \mathbb{B} \}
\]

The operator \(\equiv\) denotes equality of both arguments.

The greatest lower bound operator can be generalised to the greatest lower bound of a set of types.

\[
\bigcap_{i \in 1..n} O_i \equiv (O_1 \cap (O_2 \cap (\ldots \cap O_n)))
\]

The generalised greatest lower bound of an empty set is \(\text{Top}\). \(\text{Top}\) contains all objects.

**Definition 4.14 (Least Upper Bound)** The least upper bound of object types \(\alpha\) and \(\beta\), \((\alpha \sqcup \beta)\), is the least type that is a supertype of both.

\[
\alpha \triangleleft (\alpha \sqcup \beta) \quad \beta \triangleleft (\alpha \sqcup \beta) \quad \forall \gamma \bullet (\alpha \triangleleft \gamma \land \beta \triangleleft \gamma) \Rightarrow (\alpha \sqcup \beta) \triangleleft \gamma
\]
It is syntactically calculated for object types by taking those labels common to both. For each common label, the resulting type is the least upper bound of the associated types of the identifier.

\[(\text{Obj } F \sqcup \text{Obj } G) \equiv \text{Obj } \{ \ l \in (\text{dom}(F) \cap \text{dom}(G)) \mid l : (F(l) \sqcup G(l)) \} \]

The least upper bound of different basic types is \( \text{Top} \). For example \((\mathbb{B} \sqcup \mathbb{N})\) is \( \text{Top} \). The least upper bound of the same basic type is that same type. For example \( (\mathbb{N} \sqcup \mathbb{N}) \) is \( \mathbb{N} \). The least upper bound operator can be generalised to the least upper bound of a set of types.

\[\bigsqcup_{i \in 1..n} O_i \cong (O_1 \sqcup (O_2 \sqcup (\ldots \sqcup O_n)))\]

The generalised least upper bound of the empty set is undefined.

### 4.3 Predicate Transformer Foundation

This section provides models for states, predicates, and predicate transformers (Sections 4.3.1, 4.3.2, and 4.3.3 respectively). The connectives for predicates and predicate transformers permit the combination of heterogeneously typed predicates and predicate transformers, respectively. Section 4.3.4 explores the effects of subsumption on the monotonic subclass of predicate transformers: statements. Finally, Section 4.3.5 presents a novel definition of refinement that permits the refinement of heterogeneously typed statements.

#### 4.3.1 State Modelling

This section provides an abstract model for states and identifies the subtyping behaviour required of states. States are collections of values in which each value is associated with an identifier. A state type is a collection of identifiers, each of which is associated with a type. The syntax for construction of state types is set notation subscripted with \( ST \). Given a function \( \alpha \) which maps labels to types, the following asserts a well-formed state type.

\[
\alpha_{st} = \forall (l \in \text{dom}(\alpha)) \bullet \alpha(l)
\]

State instances are explicitly described using a function \( s \), which maps labels to their values. The function is subscripted with \( S \) to denote a state instance. The construct \( s \) is a well-formed state that has type \( \alpha_{st} \) provided its attributes are well-formed:

\[
\forall (l \in \text{dom}(\alpha)) \bullet s(l) : \alpha(l) \quad \alpha_{st} \quad \text{dom}(s) = \text{dom}(\alpha)
\]

\[s_s : \alpha_{st}\]

The empty state type, \( \top_{st} \), possesses no identifiers. Every state instance is, via subsumption, a member of the empty state type.
States subtype so that ‘larger’ states (states with more identifiers) are subtypes of ‘smaller’ states, and the types of state elements may vary covariantly (or monotonically) provided the types of ‘new’ state components are well-formed. These requirements are expressed formally in Desideratum 4.15.

**Desideratum 4.15 (Sub State Type)** Given state types $\alpha_{ST}$ and $\beta_{ST}$:

$$\text{dom}(\beta) \subseteq \text{dom}(\alpha)$$

for all $(i \in \text{dom}(\beta)) \bullet \alpha(i) \preceq \beta(i)$

for all $(i \in \text{dom}(\beta \preceq \alpha)) \bullet \alpha(i)$

$\alpha_{ST} \preceq \beta_{ST}$

This property is consistent with that used by Sekerinski [Sek96].

Having stated the desired properties for state types, a model that upholds these properties should be chosen. One possible approach is to model states as objects. However, such a model is not consistent with Desideratum 4.15 as the attributes of objects (as defined in Section 4.1) are invariantly typed. That is, given an object type, the types of the attributes of a supertype (or subtype) must be the same. In contrast, Desideratum 4.15 permits the attributes of states to vary covariantly. Extensible, covariantly annotated object types, as introduced in Chapter 5, would be a sufficient model for state types. However, records (which are extensible and whose components subtype covariantly) form a recognisable type and are therefore suggested as an alternative, simpler model. Desideratum 4.15 is shown to hold (on page 193) using the definition of states: Definition 4.16.

**Definition 4.16 (States as Records)**

$$\{j \in 1..n \bullet i_j : OT_j\}_{ST} \equiv \{j \in 1..n \bullet i_j : OT_j\}_{ST}$$

States can be constructed, extended, updated, or components selected using the corresponding record operation upon the state. The notations for the state operations are consistent with those for record operations. State cut is equivalent to record cut—it removes an element from a state. For example:

$$\{a = v, b = w\}_s \backslash_s \{a\} \equiv \{b = w\}_s$$

State union is defined as record override on the proviso that states have an empty syntactic intersection. For example, since $\{a = v\}_s$ and $\{b = w\}_s$ have no identifiers in common, their state union is justified and is calculated using record override:

$$\{a = v\}_s \cup_s \{b = w\}_s \equiv \{a = v\}_s \oplus \{b = w\}_s \equiv \{a = v, b = w\}_s$$

The object calculus provides an equivalence relation $=_{\alpha}$ for comparing terms under any given type $\alpha$. The relation has the property that any two terms equivalent under one type are also equivalent under a supertype.

The greater lower bound and least upper bound of state types can be syntactically calculated in a similar manner to that used for object types.
Subsumption of States  Like objects, a state does not alter when it is subsumed. Instead, the static information known about which identifiers are in a state is reduced. Subsumption of states means that a (‘larger’) state can be used in a situation where a (‘smaller’) state with a subset of the former’s identifiers is required. For example, a function which takes states with the identifier $a$ can also take a state with identifiers $a$ and $b$—the function disregards knowledge about $b$. Analogously, if a state function returns a state with identifiers $a$ and $b$, that same state can be regarded as a state containing only $a$—knowledge of $b$ has been disregarded.

4.3.2 Predicates

This section provides a definition of predicates and also provides novel definitions of connectives for joining heterogeneously typed predicates.

**Definition 4.17 (Predicates)**  Predicates are defined as functions from states to booleans. Given state type $\alpha$ then

$$
\text{Pred } \alpha \triangleq \alpha \to \mathbb{B}
$$

For well-formed state type $\alpha$, the predicate type $\text{Pred } \alpha$ is well-formed.

$$
\frac{}{\text{Pred } \alpha}
$$

The predicate type on the empty state type, $\text{Pred } \top_{ST}$, contains only two predicates, $\text{True}$ and $\text{False}$ (see Definition 4.19). $\text{True}$ and $\text{False}$ are also predicates on ‘larger’ state types. The following example illustrates this idea further. The predicate $a = 1$ on predicate type $\text{Pred } \{a : \mathbb{N}\}_{ST}$ constrains $a$ to 1. By subsumption, the predicate $a = 1$ also types as $\text{Pred } \{a : \mathbb{N}, b : \mathbb{N}\}_{ST}$. In the subsumed context, the predicate $a = 1$ constrains $b$ to be a natural number. Again, by subsumption, the original predicate ($a = 1$) also types as $\text{Pred } \{a : \mathbb{N}, b : \mathbb{Z}\}_{ST}$ thereby constraining $b$ to be an integer. The unsubsumed predicate $a = 1$ on type $\text{Pred } \{a : \mathbb{N}\}_{ST}$ is a predicate that constrains $a$ to 1 yet allows $b$ (and all other identifiers) to be of any value and type.

**Theorem 4.18 (Sub Predicate)**  Proof on page 194  The subtyping rule for predicates is

$$
\frac{}{\text{Pred } \beta \preceq \text{Pred } \alpha}
$$

That is, predicates subtype when the states on which they are defined vary contravariantly.
**Definition 4.19 (Predicate False)** The false predicate is defined as the lifted version of the Boolean false.

\[
\text{False} \triangleq \lambda s : T_{st} \bullet \text{false}
\]

The domain of \(\text{False}\) is the empty state type. All state types are subtypes of the empty state type. The range of \(\text{False}\) is simply the Boolean false value. Consequently, this is a function that maps all states to \(\text{false}\).

\[\therefore\]

**Theorem 4.20 (Val False)**  
*Proof on page 194*  
For every well-formed state type \(\alpha\), \(\text{False}\) can be subsumed to a predicate on that state type.

\[
\alpha \\
\text{False} : \text{Pred} \alpha
\]

The predicate \(\text{True}\) can be constructed analogously.

The Boolean logical connectives, \(\Rightarrow, \land, \lor, \lnot\) can be used for pointwise extensions on states to define corresponding predicate connectives. The definition of predicate implication is provided as an example.

**Definition 4.21 (Predicate Implication)** Given state types \(\alpha\) and \(\beta\), predicate implication is defined as the lifting of boolean implication on the greatest lower bounds of these types.

\[
p_1 : \text{Pred} \alpha \quad p_2 : \text{Pred} \beta \quad (\alpha \sqcap \beta) \\
\therefore p_1 \Rightarrow p_2 \triangleq \lambda s : (\alpha \sqcap \beta) \bullet p_1(s) \Rightarrow p_2(s)
\]

**Theorem 4.22 (Val Predicate Implication)**  
*Proof on page 194*  
The type of a predicate implication is the greatest lower bound of the state types of the two operands, provided that type is well-formed.

\[
p_1 : \text{Pred} \alpha \quad p_2 : \text{Pred} \beta \quad (\alpha \sqcap \beta) \\
\therefore (p_1 \Rightarrow p_2) : \text{Pred} (\alpha \sqcap \beta)
\]

For example, given \((a = 1) : \text{Pred} \{a : \mathbb{N}\}_{st}\) and \((b = 1) : \text{Pred} \{b : \mathbb{N}\}_{st}\), then

\[a = 1 \Rightarrow b = 1\]

is a predicate on the type \(\text{Pred} \{a : \mathbb{N}, b : \mathbb{N}\}_{st}\). The definitions of predicate conjunction and predicate disjunction are analogous.

Predicate validity is used to determine whether the predicate is true for all states on which it is defined.
**Definition 4.23 (Validity)** Given state type $\alpha$ and predicate $p$, validity is defined as:

$$p : \text{Pred } \alpha$$

$$\forall s : \alpha \bullet p(s)$$

**Definition 4.24 (Entailment)** Entailment is the validation of predicate implication. For predicates $p$ and $q$ on state types $\alpha$ and $\beta$ respectively, entailment is defined as:

$$p : \text{Pred } \alpha \quad q : \text{Pred } \beta \quad (\alpha \sqcap \beta)$$

$$p \Rightarrow q \triangleq [p \Rightarrow q] (\alpha \sqcap \beta)$$

For the calculus to be complete, the traditional properties of predicates, e.g., idempotence, distributivity, and de Morgan, must be shown to hold. In an untyped predicate calculus, this can be shown by proving that the predicates form a complete Boolean lattice under implication (refer to Appendix C and [BvW98, Chapter 2]). The traditional properties follow from this. In this typed predicate calculus the proofs are similar. The main impediment to the proofs is that not all meets (conjunctions) of predicates exist. Informally, in the untyped predicate calculus the predicate $s = 1 \land s = \text{false}$ is equivalent to $\text{False}$. However, in this typed predicate calculus, the predicate $s = 1 \land s = \text{false}$ has no semantics as the meet $\bigwedge (\mathbb{N} \cap \mathbb{B})$ is undefined. This is not an issue in practice as typechecking disallows this form of predicate.

### 4.3.3 Predicate Transformers

This section defines predicate transformers and explores their subtyping behaviours. Predicate transformers are defined as functions from postcondition predicates to precondition predicates.

**Definition 4.25 (Predicate Transformers)** For state types $\alpha_{\text{Post}}$ and $\alpha_{\text{Pre}}$:

$$P\text{trans } \alpha_{\text{Post}} \alpha_{\text{Pre}} \triangleq \text{Pred } \alpha_{\text{Post}} \rightarrow \text{Pred } \alpha_{\text{Pre}}$$

To allow piecewise refinement of programs, it is necessary to only use monotonic predicate transformers within a refinement calculus. The property of monotonicity is discussed further on page 11. Predicates transformers are pragmatically constrained to those that are monotonic. The statements that are introduced in the next section are defined to be monotonic.

**Theorem 4.26 (Sub Predicate Transformers)** Proof on page 195 Predicate transformers subtype when the postcondition state varies covariantly and the precondition state varies contravariantly.

$$\alpha \preceq \beta \quad \gamma \preceq \delta$$

$$P\text{trans } \alpha \delta \preceq P\text{trans } \beta \gamma$$

as $\text{Pred } \beta \preceq \text{Pred } \alpha$ and $\text{Pred } \delta \preceq \text{Pred } \gamma$. 
CHAPTER 4. A TYPED REFINEMENT CALCULUS

Table 4.1: Statement Definitions

<table>
<thead>
<tr>
<th>Statement</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>abort_α</td>
<td>( \lambda p : \text{Pred} \alpha \bullet \text{False} )</td>
</tr>
<tr>
<td>magic_α</td>
<td>( \lambda p : \text{Pred} \alpha \bullet \text{True} )</td>
</tr>
<tr>
<td>skip_α</td>
<td>( \lambda p : \text{Pred} \alpha \bullet p )</td>
</tr>
<tr>
<td>{p}_α</td>
<td>( \lambda q : \text{Pred} \alpha \bullet p \land q )</td>
</tr>
<tr>
<td>[p]_α</td>
<td>( \lambda q : \text{Pred} \alpha \bullet p \Rightarrow q )</td>
</tr>
<tr>
<td>\langle st\rangle_β</td>
<td>( \lambda p : \text{Pred} \beta \bullet (\lambda s : \alpha \bullet p(st)) )</td>
</tr>
<tr>
<td>(i₁,...,iₘ := e₁,...,eₘ)_α</td>
<td>( (\lambda p : \text{Pred} \alpha \bullet p[i₁,...,iₘ\setminus e₁,...,eₘ])_α )</td>
</tr>
<tr>
<td>enter v : V := initv_α</td>
<td>( \langle \lambda s : \alpha \bullet s \cup { v = \text{eval initv} } \rangle_α )</td>
</tr>
<tr>
<td>exit v : V_α</td>
<td>( \langle \lambda s : (\alpha \cap { v : V })_α \bullet (\cap t : \alpha \mid t = s \bullet t) \rangle_α )</td>
</tr>
<tr>
<td>v : [pre, post]_α</td>
<td>( \lambda p : \text{Pred} \alpha \bullet pre \land (\forall v \bullet post \Rightarrow p)[v₀ \setminus v] )</td>
</tr>
<tr>
<td>(pt₁ \cap pt₂) ≡ \lambda p : \text{Pred} \beta \bullet pt₁(p₂p)</td>
<td></td>
</tr>
<tr>
<td>pt₁ \cap pt₂ ≡ \lambda p : \text{Pred} (\alpha \cup \beta) \bullet pt₁ p \land pt₂ p</td>
<td></td>
</tr>
<tr>
<td>[ \begin{array}{c} \text{var } a : A := av \bullet P \end{array} ]_α</td>
<td>( \lambda \text{post : Pred } \alpha \bullet (\exists lcon \bullet pt \text{ post}) )</td>
</tr>
</tbody>
</table>

For instance, the predicate transformer \( (a, b := 1, 1) : \text{Ptrans} \{ a, b : \mathbb{Z} \}_\alpha \{ a, b : \mathbb{Z} \}_\beta \) guarantees postcondition \( a = 1 \land b = 1 \) with a precondition \text{True} on a state containing elements \( a \) and \( b \) (\text{Pred} \{ a, b : \mathbb{Z} \}_\alpha \). The assignment can also be applied to a postcondition of type \text{Pred} \{ a : \mathbb{Z} \}_\alpha. For example, it can be used to establish the predicate \( a = 1 \) of type \text{Pred} \{ a : \mathbb{Z} \}_\alpha. In a similar fashion, under subsumption the predicate transformer \( a, b := 1, 1 \) may guarantee \( a = 1 \land b = 1 \) with a precondition \text{True} on a state type consisting of \( a, b \) and \( c : \mathbb{Z} \), i.e., \text{Pred} \{ a, b, c : \mathbb{Z} \}_\alpha. Alternatively, the predicate transformer could have been subsumed to have a precondition state type where \( c \) was a Boolean (\text{Pred} \{ a, b : \mathbb{Z}, c : \mathbb{B} \}_\alpha). Applying a postcondition predicate of type \text{Pred} \{ a, b : \mathbb{Z}, c : \mathbb{B} \}_\alpha to the assignment is not well typed as the assignment does not subsume to the type \text{Ptrans} \{ a, b : \mathbb{Z}, c : \mathbb{B} \}_\alpha \{ a, b : \mathbb{Z} \}_\alpha.

4.3.4 Statements

Statements are monotonic predicate transformers\(^3\). The abort statement and demonic choice operators are examined here to illustrate the effects of typing the predicate transformers. Other statements used in this thesis are listed in Figure 4.1 and presented more thoroughly in Appendix B.3.

\(^2\)The subscript denoting the type of an assignment is dropped when obvious from the context.

\(^3\)Refer to page 11 for a discussion of monotonicity in the context of the refinement calculus.
Definition 4.27 (Abort) The abort predicate transformer always returns the predicate False. Typing affects the abort statement by forcing it to be parameterised on the postcondition state—forming a family of abort statements. This parameter is omitted when it is obvious from the context.

\[ \text{abort}_\alpha \triangleq \lambda p : \text{Pred} \alpha \bullet \text{False} \]

Example 4.28 (Subsuming an Abort) The difference between members of the abort family can be distinguished by an example in which one is subsumed. For instance, \( \text{abort}_{\{a : N, b : N\}_ST} \) is a predicate transformer that maps all predicates of type \( \text{Pred} \{a : N, b : N\}_ST \) (and through subsumption, predicates of types \( \text{Pred} \{a : N\}_ST \), \( \text{Pred} \{b : N\}_ST \) and \( \text{Pred} \top_{ST} \)) to the predicate False. However, \( \text{abort}_{\{a : N\}_ST} \) only maps predicates of types \( \text{Pred} \{a : N\}_ST \) and \( \text{Pred} \top_{ST} \) to False. Hence, \( \text{abort}_{\{a : N, b : N\}_ST} \) can be used wherever \( \text{abort}_{\{a : N\}_ST} \) is used and it will produce the same effects but not vice versa. For \( PT \triangleq \text{Ptrans} \{a : N\}_ST \top_{ST} \)

\[ \text{abort}_{\{a : N, b : N\}_ST} = PT \text{ abort}_{\{a : N\}_ST} \]

Theorem 4.29 (Val Abort) Proof on page 195 Given state type \( \alpha \), the abort predicate transformer types as \( \text{Ptrans} \alpha \top_{ST} \).

\[ \text{abort}_\alpha : \text{Ptrans} \alpha \top_{ST} \]

Given state type \( \alpha \), the magic predicate transformer \( \text{magic}_\alpha \) can be constructed in an analogous manner.

Definition 4.30 (Demonic Choice) The classical definition of demonic choice (predicate transformer meet) is the lifting of predicate conjunction. Given predicate transformers \( pt_1 \) and \( pt_2 \), demonic choice is defined when the greatest lower bound of the precondition state types of \( pt_1 \) and \( pt_2 \) is well-formed. This restriction ensures that all weakest preconditions of \( pt_1 \) can be conjoined with those of \( pt_2 \).

\[ pt_1 : \text{Ptrans} \alpha \delta \quad pt_2 : \text{Ptrans} \beta \gamma \quad (\delta \sqcap \gamma) \]

\[ pt_1 \sqcap pt_2 \triangleq \lambda p : \text{Pred} (\alpha \sqcup \beta) \bullet pt_1 p \land pt_2 p \]

Theorem 4.31 (Val Demonic Choice) Proof on page 196 Demonic choice types as a predicate transformer from the join of the state types of the postconditions to the meet of the state types of the preconditions provided the meet is well-formed.

\[ pt_1 : \text{Ptrans} \alpha \delta \quad pt_2 : \text{Ptrans} \beta \gamma \quad (\delta \sqcap \gamma) \]

\[ (pt_1 \sqcap pt_2) : \text{Ptrans} (\alpha \sqcup \beta) (\delta \sqcap \gamma) \]
4.3.5 Statement Refinements

Refinement, the predicate transformer ordering relation, is defined via predicate entailment. Classically, the refinement relation is used to ‘compare’ two statements of the same type. Following is an innovative refinement relation that allows ‘comparisons’ between statements of heterogeneous types.

**Definition 4.32 (Refinement)** Given predicate transformers \( pt_1 \) and \( pt_2 \), refinement between them is defined as:

\[
\begin{align*}
pt_1 : \perp \text{Ptrans } \alpha \delta \quad pt_2 : \text{Ptrans } \beta \gamma \quad \beta \leq \alpha \quad (\delta \cap \gamma) \\
\circ\quad pt_1 \sqsubseteq pt_2 \iff \forall p : \text{Pred } \alpha \bullet pt_1 p \Rightarrow pt_2 p
\end{align*}
\]

The premises \( \beta \leq \alpha \) and \( (\delta \cap \gamma) \) are the minimum typing requirements for the definition to be well-formed. Since the goal of using refinement is to replace the original code with the refined code the definition may be more general than is required: the subsumption of \( pt_2 \) to \( pt_1 \), that is, \( \text{Ptrans } \beta \gamma \leq \text{Ptrans } \alpha \delta \) (alternatively \( \beta \leq \alpha \land \delta \leq \gamma \)), may be sufficient.

The ‘most defined type’ premise, as introduced on page 37,

\[
pt_1 : \perp \text{Ptrans } \alpha \delta
\]

ensures that \( \text{Ptrans } \alpha \delta \) is the declaration type of \( pt_1 \) and not a type obtained by subsumption. That is, there is no subtype of \( \text{Ptrans } \alpha \delta \) that \( pt_1 \) is an element of. The ‘most defined type’ property is static as opposed to a type check which is evaluated dynamically. It is used to constrain the declaration type of an object, not to regain attributes lost due to subsumption. The ‘most defined type’ constraint is applied to prevent the subsumption of \( pt_1 \) to, for example, the type \( \text{Ptrans } \top_{\mathcal{ST}} \delta \). If this were allowed, refinement could be reduced to the following:

\[
pt_1 \sqsubseteq pt_2 \equiv pt_1(\text{True}) \Rightarrow pt_2(\text{True}) \land pt_1(\text{False}) \Rightarrow pt_2(\text{False})
\]

as the predicate on state type \( \top_{\mathcal{ST}} \) only contains \( \text{True} \) and \( \text{False} \). This definition would allow the refinement of, for example, \( a := 2 \) to \( a := 3 \).

**Example 4.33 (Heterogeneously Typed Statement Refinement)** The definition of refinement allows statements that modify variables outside the state to act in lieu of statements that only modify state variables. For example, in an environment consisting of only variables \( a \) and \( b \), the statement \( a, b, c := a + 1, b + 1, 1 \) is a refinement of the statement \( a, b := a + 1, b + 1 \). Similarly the statement \( a, b, c := a + 1, b + 1, \text{false} \) is also a refinement. These refinements are intuitive: since \( c \) is not in the state, the correctness of the program does not depend upon the value of \( c \). Consequently, \( c \) can be arbitrarily modified without violating the program’s correctness. This is an open world view\(^4\) [Mah99, p90]. In an open world, variables outside the state may be modified. In a closed world, variables outside the state may not be modified.

\[^4\]The phrase “open world view” was coined by Utting[Utt92].
Given that the refinement relation is defined under an open world view, the frame of a specification can be expanded with variables outside the specification’s state.

**Theorem 4.34 (Open World Specification)**  **Proof on page 199** In an environment consisting only of state variables $\vec{z}$, and which is disjoint from $\vec{w}$:

$$\vec{z} : [pre, post]_{\{\vec{z}\}_{ST}} \sqsubseteq \vec{z}, \vec{w} : [pre, post]_{\{\vec{z}, \vec{w}\}_{ST}}$$

This frame extension enlarges the specification’s state environment. The state of the specification must be reduced to that of the original specification before it can be used in lieu of the original. In the classical refinement calculus this is achieved by encapsulating it with a variable block scope. The Introduce Local Variable Block (B.55) rule is used to accomplish this. In this thesis, subsumption can also be used to reduce the specification’s type to that of the original specification.

To complete this refinement calculus, the traditional properties of refinement calculi need to be reproved. Since the main change to the definitions is the addition of type information, the proofs are similar. The main impediment of the reproof of the traditional predicate calculus properties for a typed calculus was that the conjunction of some predicates does not have a semantics. A similar problem exists when reproving the traditional refinement calculus properties for a typed calculus. That is, the meet of some heterogeneously typed predicate transformers does not have a semantics. As discussed previously for the conjunction of some predicates, typechecking avoids this lack of semantics in practice.

### 4.4 Summary

This chapter has illustrated the addition of type information to classical refinement calculus definitions. The purpose of these modifications is to allow objects to be typed when predicate transformers are embedded within them. One innovation of the chapter is the definition of a refinement relation that handles heterogeneously typed statements. This has interesting implications. For instance, it allows the statement $a, b := a + 1, b + 1$ in an environment consisting only of variables $a$ and $b$ to be refined by the statement $a, b, c := a + 1, b + 1, false$. A similar property is used in Chapter 7 to support the addition of attributes to objects.

The foundation presented here is used in Chapter 5 to provide an example representation of objects with embedded predicate transformers. It is also used in Chapter 6 as the basis for an object-oriented language. That language is extended to an object-oriented refinement calculus in Chapter 7.
Chapter 5

An Object Representation

Using the typed predicate transformers developed in Chapter 4, an original representation for objects is presented that supports binary methods (introduced on page 17) and embedded predicate transformers. The main result of the chapter is the proof that predicate transformers, when embedded into the object representation, subtype appropriately (as defined by Desideratum 5.2).

It is emphasised that the model presented here is only one possible model. The proposed model suffers from the lack of support for unbounded nondeterminism. The lack of support for unbounded nondeterminism is not of practical significance, however, as language implementations of types are finite. The remainder of the thesis uses an abstraction of the desired object model. Consequently, an alternative model that does not suffer this problem can be used to replace the current model and all properties and techniques of the thesis could be carried across. This abstraction is one of the main strengths of the thesis.

The object representation presented here is based on the object calculus $\text{FOb}_{\approx_{\mu}}$ as introduced in Chapters 2 and 4. A full axiomatisation of $\text{FOb}_{\approx_{\mu}}$ is presented in Appendix A. The object representation supports binary methods and the ability to add methods to subclasses, while maintaining a general rule of subsumption. Late binding can be modelled with this object representation by invocation of the embedded method. Section 5.1 motivates the object representation by identifying the necessary typing behaviour of objects with embedded methods and the lack of success using $\lambda$-calculus-based semantics. Section 5.2 introduces the object calculus concepts used to construct the model. Based on these concepts, Section 5.3 presents several object representation models. Some of these models are not sufficient as they do not uphold the typing and subtyping properties discussed in Section 5.1. The reasons for these violations are discussed. The final model, however, is proven to be sufficient.

5.1 Motivation

Choosing an object representation is fundamental to the semantics of an object-oriented refinement calculus. Chapter 3 presents several examples of object representations in ex-
existing object-oriented refinement calculi. To simplify the typing system, the majority of those approaches have object representations that decouple fields from methods. This approach, however, forces a late binding mechanism to be added as an extra feature to the calculus. Examples of such a late binding mechanism are the use of sum types [MS97] or Utting's function [Utt92]. This late binding mechanism adds an additional layer of complexity to the semantics of the resulting object-oriented refinement calculus. In contrast, given an appropriate object representation in which methods are embedded, it is possible to model late binding as the execution of the contained method. Given that the simplification of the typing system is the main reason provided for abandoning the embedded approach [BMvW00, p35], it would seem more productive to search for a simple method-embedded object type rather than to complicate the semantics.

Many researchers have investigated the types required for modelling objects. Palsberg and Schwartzbach [PS94] present an introduction to the problems of object typing and the solutions that current object-oriented languages employ. They suggest that the lack of maturity of types for objects stems from the origins of type theoretic research as a discipline of logic—remote from the types required for practical object-oriented programming languages. It is evident that the $\lambda$-calculus alone is not sufficient for practical object-oriented language semantics [AC96, Section 6.7, Chapter 18] [PS94, Section 2.6] as it does not have the "desired modelling power" [AC96, p51]. Informally, if object-oriented languages could be reduced to procedural languages then $\lambda$-calculi would be sufficient.

**Example 5.1 (Self-Application Semantics)** One example of a $\lambda$-calculus-based object-oriented semantics is the self-application semantics. This semantics maps objects to records of functions. Consider a method $f$ that has type $C \rightarrow R$ where $C$ is the method's host type and $R$ is the result type of the method. Given subtype $D$ of object $C$, that is $D \subseteq C$, then $f$ is also of type $D \rightarrow R$ as $f$ accepts arguments of type $D$ as they have type $C$ by subsumption\(^1\). By this argument it can be deduced that function types are contravariant in their left argument: $D \subseteq C \Rightarrow C \rightarrow R \subseteq D \rightarrow R$. Unfortunately, the opposite is required to model object types. Given an object of type $D$, by subsumption, it should also be of type $C$. So its methods, of type $D \rightarrow R$, should, by subsumption, be of type $C \rightarrow R$—thus requiring function types to be covariant in their left argument, contradicting the original analysis. Similar problems occur for other encodings of objects using $\lambda$-calculi.

\(~\)\(^1\)See the work of Abadi and Cardelli [AC96, p21,p76] for further discussion.

To solve the typing difficulties of object orientation, Abadi and Cardelli [AC96] investigated object calculi. Amongst other successes, their research provides types for objects with methods that return other objects of the same type (the *Self* type). To solve technical problems they eventually [AC96, Chapter 16] abandon standard denotational models and
provide an object type that is shown consistent by means of an operational semantics. Despite this extensive work, types for objects that possess binary methods and support subsumption still do not exist. This causes difficulties for the integration of the refinement calculus with the object-oriented programming paradigm. An intuitive, abstract model of (refinement calculus) objects is one in which objects contain predicate transformers as methods that manipulate the host. Since the objects manipulate themselves, a variable of the host type, $O$, must occur in both the methods’ precondition and postcondition state types. That is, the methods must be of the type $\text{Pred } \alpha \{O\} \rightarrow \text{Pred } \beta \{O\}$ where $\alpha \{O\}$ denotes the possible occurrence of $O$ in state type $\alpha$. Using $O$ in both the precondition and postcondition means that the predicate transformers are binary methods. A model is provided here that solves the binary method problem for the specific case of embedded predicate transformers. The rule for subtyping these objects is provided along with its proof. Restrictions are placed on this model to ensure that it is consistent with a standard denotational semantics.

### 5.2 Additional Object Calculus Concepts

This section introduces several object calculus concepts that are used later for the presentation of models of (object-oriented refinement calculus) objects. These concepts are variance annotations, recursive object types, dynamic type checking, and covariant self types.

#### 5.2.1 Variance Annotations

Given object types $C$ and $D$ (as introduced in Section 4.1), for $D$ to be a subtype of $C$ ($D \preceq C$) the types of the attributes must be invariant—the same (though $D$ may have more attributes than $C$). To provide object types with variant attribute types, Abadi and Cardelli introduce variance annotations: invariance ($\circ$), covariance ($+$) and contravariance ($-$). Omitted annotations are, by convention, invariant. An invariance annotation ($\circ$) constrains the annotated field type to be invariant. Annotating an attribute with the covariant syntax ‘$+$’ allows the attribute to subtype covariantly (or monotonically). For example, for types $C$ and $D$ where $D \preceq C$ then

$$\text{Obj } \{ \text{l}^+ : D \} \preceq \text{Obj } \{ \text{l}^+ : C \}$$

To prevent an unsound typing system, Abadi and Cardelli [AC96] restrict covariant attributes such that they are permitted to be updated only by the host, i.e., they can be selected but not updated by an external client. Annotating an attribute with the contravariant syntax ‘$-$’ allows contravariant (anti-monotonic) subtyping.

$$\text{Obj } \{ \text{l}^- : C \} \preceq \text{Obj } \{ \text{l}^- : D \}$$

---

2 A similar approach is followed by Palsberg and Schwartzbach [PS94, p26].

3 As illustrated by [AC96, p109].
Abadi and Cardelli restrict contravariant attributes such that they can be selected only from within the host, i.e., they can be updated but not selected by an external client.

An interpretation of variance annotations in terms of the underlying object calculus is presented within Appendix A.2.

### 5.2.2 Recursive Object Types

With the object types of Section 4.1, it is not possible to type an object that contains a method that returns an object of its host’s type. Recursive types [AC96, Chapter 9], however, provide this facility. The recursive object type $\mu(X)\text{Obj} \ C\{X\}$ is the unique solution to $X = \text{Obj} \ C\{X\}$ where $C$ is the ‘desired’ object type with recursive references to itself. The syntax $C\{X\}$ means that the self type $X$ can occur freely, or unbound, within $C$. In a similar fashion to unfolding a recursive procedure, the type $\mu(X)\text{Obj} \ C\{X\}$ can be unfolded by replacing the bound $X$ with the body $\mu(X)\text{Obj} \ C\{X\}$ to achieve the type $\text{Obj} \ C\{X\} \setminus \mu(X)\text{Obj} \ C\{X\}$ where $C\{X\} \setminus Y$ denotes the syntactic substitution of $X$ with $Y$ within $C$. The type $\mu(X)\text{Obj} \ C\{X\}$ and its unfolding $\text{Obj} \ C\{X\} \setminus \mu(X)\text{Obj} \ C\{X\}$ are isomorphic yet not equivalent. Thus instances of the former are not instances of the latter, and vice versa. To relate instances of these types, the fold and unfold constructs are used. For example, given the recursive type $A \triangleq \mu(X)\text{Obj} \{ \text{data} : \mathbb{N}, \text{next} : X \}$ and an instance $a$ of that type, then unfolding $a$ produces an isomorphic object of the unfolded type:

$$\text{unfold}(a) : \text{Obj} \{ \text{data} : \mathbb{N}, \text{next} : \mu(X)\text{Obj} \{ \text{data} : \mathbb{N}, \text{next} : X \} \}$$

Conversely, folding an instance of the unfolded type leads to an isomorphic instance of the original type:

$$\text{fold}(A, \text{unfold}(a)) : A$$

### 5.2.3 Dynamic Type Checking

When an object is subsumed the object does not change, however, statically less information is known about its attributes. Even though static typing information may be lost during subsumption, the ‘lost’ attributes may still be utilised through late binding. In a pure object-oriented language, dynamic dispatch is the only mechanism for accessing attributes ‘lost’ by subsumption. Most languages, however, provide a facility for regaining direct access to lost attributes by examining the run-time type of objects. This feature is known as the type-case construct. It is also known as dynamic type check and type casting. Abadi and Cardelli introduce the following syntax for dynamic type checking:

$$\text{typecase } a \mid (x : A)d_1 \mid d_2$$

When $a$ is of dynamic type $A$, it is bound to $x$ and $d_1$ is returned, otherwise $d_2$ is returned.
It is possible to use dynamic type checks in a methodologically unsound manner. They can violate object encapsulation, allowing access to private attributes. Dynamic type checks can also reduce the extensibility of the code. When a new subclass is introduced, previous uses of type-cases may need to be extended. This violates the purist principle of object orientation that the addition of a new class does not require the recoding of existing classes. These methodological issues require that dynamic type checks be used judiciously. Much of the work on object-oriented types has focused on reducing the need for dynamic type checks. However, dynamic type checking is currently the only mechanism which supports binary methods.

5.2.4 Self Types

Self types [AC96, Chapter 15] have been investigated as part of the effort to reduce the reliance upon dynamic type checks. Consider the following class where an instance of the class type is returned from one of the class’s methods (m):

```plaintext
class C is
  var x : Z := 0;
  method m() : C is body end;
end
```

where body returns an instance of C. Given the following subclass D

```plaintext
subclass D of C is
  var y : Z := 0;
end
```

and an instance d of D, then in general it is unsound to type the result of method m of d as D as m may return an arbitrary instance of type C that is not of type D, e.g., one without attribute y. However, if m is restricted such that it may only return a modification of the host (or self) then it would be sound to type the inherited method as D. The type Self is introduced to allow this more precise typing, and hence reduce the need for subsequent dynamic type checks. The original class would be written with the result types replaced by the type Self:

```plaintext
class C is
  var x : Z := 0;
  method m() : Self is body end;
end
```

No longer can body return an arbitrary instance: it must return an object of exactly the same type as the host. Hence, for the (new) subclass D, the method m must return an instance of D.

---

4This example is derived from the work of Abadi and Cardelli [AC96, p23].
This example uses self types in a covariant position. Contravariant use of the self type, as exemplified in class \( E \), is unsound for subsumption. Given an instance \( f : F \) where \( F \preceq E \), then \( f \) may be subsumed to be of type \( E \), i.e., \( f : E \). Consequently the method \( m \) of \( f \) may be called with a parameter \( e \) of type \( E \). However, \( f \circ m \) will only work correctly with a parameter of type \( F \) as, in this example, it uses the field \( y \) which only occurs in instances of \( F \) [Coo89].

```plaintext
class E is
  var x : Z := 0;
  method m(other : Self) is \( \varsigma(self : Self) \) self \( \circ \) x \( \equiv \) other \( \circ \) x end;
end

class F is
  var x, y : Z := 0;
  method m(other : Self) is \( \varsigma(self : Self) \) self \( \circ \) x \( \equiv \) other \( \circ \) y end;
end
```

### 5.3 Object-Typed Models

This section presents an innovative model of an (object-oriented refinement calculus) object type that contains embedded predicate transformers as methods. Using this object type, late binding can be modelled simply by the invocation of the contained predicate transformer. Additionally, since the objects have embedded methods the model is useful for both object-based and class-based refinement calculi.

Embedding methods in objects forces the objects to be recursive as the methods are a component of the entities that they manipulate. If the object type is represented as \( X \) then an object representation must be chosen such that methods of type \( Ptrans \{ self : X \}_{st} \{ self : X \}_{st} \)

5 can be embedded and subsumed appropriately: methods of subtypes subsume to the methods of supertypes.

**Desideratum 5.2 (Object Subtyping)** For instance, given an object type with a method \( m \)

\[
X \equiv Obj \{ m : Ptrans \{ self : X \}_{st} \{ self : X \}_{st} \}
\]

and the intended ‘subtype’

\[
Y \equiv Obj \{ m : Ptrans \{ self : Y \}_{st} \{ self : Y \}_{st},
  n : Ptrans \{ self : Y \}_{st} \{ self : Y \}_{st} \}
\]

then the type of method \( m \) in \( Y \) should be a subtype of the type of method \( m \) in \( X \), informally:

\[
Ptrans \{ self : Y \}_{st} \{ self : Y \}_{st} \preceq Ptrans \{ self : X \}_{st} \{ self : X \}_{st}
\]

\footnote{Here the choice of associating the host object with the variable \( self \) is made.}
A model for (object-oriented refinement calculus) object types with embedded methods should display this subtyping behaviour.

Choosing a model that conforms to this subtyping behaviour is, however, analogous to solving the binary method problem (for this context) as the occurrence of the ‘self’ type \((X)\) in the precondition is contravariant. Current research has not found a suitable model for contravariant occurrences of ‘self.’

Recursive object types are not a sufficient model for object-oriented refinement calculus object types. Even the covariant occurrence of \(X\) in the ‘postcondition’ portion of the type causes difficulties. Consider the following recursive objects and their types.

6 Example adapted from [AC96, p123].
A semantic error would occur during execution as \( c \) does not have a \( z \) field. This problem arises as the updated \( d \) returns an object of type \( C \). One solution to the problem is to restrict \( d \) to only return modified versions of itself—not arbitrary objects. This can be achieved by covariantly annotating the methods, thus blocking the update in the previous example of the \( return\_host \) method. Consequently, the combination of recursive and variance annotated types is now discussed.

Recursive, variantly annotated object types support both covariant and contravariant occurrences of self. However, the associated subtyping rule is not strong enough to allow subtypes that also have the contravariant occurrences of self. The subtyping rule, formed by the combination of the rules for subtyping variantly annotated object types (A.39) and recursive types (Sub Rec (A.16)), is as follows [AC96, p160].

**Axiom 5.3 (General Variance Annotated Recursive Types)** Consider object types \( C \) and \( D \) which are recursive and variance-annotated. Variance annotations are denoted by \( v_i \) and \( v'_i \).

\[
C \equiv \mu(X)\text{Obj} \left\{ i \in 1..n \bullet l^i : B_i\{X\} \right\} \\
D \equiv \mu(X')\text{Obj} \left\{ i \in 1..n + m \bullet l'^i : B'_i\{X'\} \right\}
\]

\( D \) is a subtype of \( C \) if, under the assumption \( X' \preceq C \), the types of \( D \)'s attributes are subtypes of the corresponding attribute types of \( C \) where occurrences of \( X \) are replaced by \( C \).

\[
E \vdash C \quad E \vdash D \quad \text{for all } i \in 1..n \bullet \text{E} \cup \{X' \preceq C\} \vdash B'_i\{X'\}^{v'_i} \preceq (B^i\{X\} \setminus C) \\
E \vdash D \preceq C
\]

For example, \( \mu(X')\text{Obj} \left\{ l^+ : X', m^+ : X' \right\} \) is a subtype of \( \mu(X)\text{Obj} \left\{ l^+ : X \right\} \) as:

\[
X' \preceq C \vdash X'^+ \preceq (X^+)[X \setminus C]
\]

which is equivalent under the substitution to

\[
X' \preceq C \vdash X'^+ \preceq C^+
\]

which holds given Sub Covariant (A.41).

Recursive, variance-annotated types can be embedded with predicate transformers to obtain a subtyping rule identical to Desideratum 5.2 except that the type of \( self \) in the precondition of the subtype \( (D) \) is the same as the supertype \( (C) \).

**Theorem 5.4 (Variance Annotated Recursive Types)** Proof on page 59 The object type

\[
C \equiv \mu(X)\text{Obj} \left\{ m^+ : (Ptrans \{self : X\}_{sr} \{self : X\}_{sr}) \right\}
\]

is a supertype of the following object type \( D \) which extends \( C \) with a method \( n \):

\[
D \equiv \mu(X')\text{Obj} \left\{ m^+ : (Ptrans \{self : X'\}_{sr} \{self : X'\}_{sr}), \right\} \\
m^+ : (Ptrans \{self : X'\}_{sr} \{self : C\}_{sr}), \right\}
\]
CHAPTER 5. AN OBJECT REPRESENTATION

**Proof of 5.4 from p58 (Variance Annotated Recursive Types)**

Using 5.3 the proof reduces to showing the following, assuming $X' \preceq C$:

$$Ptrans \{self : X'\}_{st} \{self : C\}_{st} \preceq Ptrans \{self : C\}_{st} \{self : C\}_{st}^+$$

This can be shown using the subtyping rule of predicate transformers, 4.26, and Sub State Type (4.15).

**QED**

Using self types instead of variance-annotated, recursive types does not result in similar properties as the subtyping rule only allows occurrences of self in covariant positions.

Theorem 5.4 is not identical to Desideratum 5.2 due to the inability of the type system to deal with contravariant occurrences of self. One solution to overcome this problem is to use dynamic typing—or dynamic type checks. Care must be taken, however, to ensure that the use of the dynamic type checks avoids the methodological issues identified earlier. Dynamic type checks are now used to obtain a rule identical to Desideratum 5.2.

Consider an object instance $d$ of type $D$ (as defined in Theorem 5.4) with methods $m$ and $n$ (with method bodies $m_{pt}$ and $n_{pt}$ respectively).

$$d \equiv \text{object} \{ \text{m} = \text{m}_{pt}, \text{n} = \text{n}_{pt} \}$$

The type of each method in $D$ is $Ptrans \{self : D\}_{st} \{self : C\}_{st}^+$, as the recursive quantifier $X'$ equates to $D$. Consequently, method $m$ cannot use the attribute $n$ as $m$’s precondition type is $\text{Pred} \{self : C\}_{st}$ whereas $n$ is an attribute of $D$ but not $C$. By dissecting $m$ (and $n$) and inserting an appropriate dynamic type check, the lost attributes of the precondition are re-acquired. Given the type of $d$, i.e., $D$, it is known that $d$’s method $m_{pt}$ is equivalent to:

$$\lambda \text{post} : \text{Pred} \{self : X'\}_{st} • (\lambda \text{s} : \{self : C\}_{st} • ((\text{m}_{pt} \text{ post} \text{s})))$$

This predicate transformer is now modified to include a dynamic type check on $self$ in the precondition state $s$ to regain access to those attributes in $D$ but not in $C$. It is statically known that $s$ is of type $\{self : C\}_{st}$. The following typecase statement allows $s$ to be known dynamically to be of type $\{self : D\}_{st}$.

$$\text{typcase } s \text{ when } (t : \{self : D\}_{st})$$
$$\text{then } (\text{new}_{m_{pt}} \text{ post})(t)$$

The type-case performs a dynamic check on $s$. If $s$ is dynamically of type $\{self : D\}_{st}$ then the type-case binds $s$ to $t$ which is statically of type $\{self : D\}_{st}$. Since $t$ has regained the ‘lost’ attributes of $s$, the predicate transformer $\text{new}_{m_{pt}}$ can be written so that it has type $Ptrans \{self : D\}_{st} \{self : D\}_{st}$. The type-case does not have an else branch as it

---

7For readability, the requirement to recursively fold the object has been omitted here.
will be placed in a context where \( s \) is always of the dynamic type \( \{ \text{self} : D \}_\text{str} \). The object \( \texttt{new}_d \) uses such a type check to provide an improved version of the object \( d \).

\[
\text{new}_d \triangleq \begin{cases} 
    m & \text{typecase } s \text{ when } (t : \{ \text{self} : D \}_\text{str}) \\
    \text{then } (\texttt{new}_\text{mu} \ post)(t), \\
    n & = \ldots 
\end{cases}
\]

Dynamically \( \texttt{new}_d \) acts as though it is of the desired type:

\[
\mu(X')\text{Obj} \left\{ 
    m : \text{Ptrans} \{ \text{self} : X' \}_\text{str} \{ \text{self} : X' \}_\text{str}^+, \\
    n : \text{Ptrans} \{ \text{self} : X' \}_\text{str} \{ \text{self} : X' \}_\text{str}^+
\right\}
\]

though statically (for subtyping purposes) it is of type \( D \):

\[
\mu(X')\text{Obj} \left\{ 
    m : \text{Ptrans} \{ \text{self} : X' \}_\text{str} \{ \text{self} : C \}_\text{str}^+, \\
    n : \text{Ptrans} \{ \text{self} : X' \}_\text{str} \{ \text{self} : C \}_\text{str}^+
\right\}
\]

Recursive, variance-annotated object types with ‘precondition’ dynamic type checks have been shown to possess the typing properties desired for objects with embedded predicate transformers. For this reason, an object calculus with support of these features, namely \( \text{FOb}_{\text{sl}} \), was chosen as the foundation. These object types are not without limitation however. The semantics of the object calculus is based on a variant of the self-application denotational semantics [AC96, p185]. This semantics uses a continuous function space, i.e., functions that are constrained by continuity [MV94]. Dijsktra [Dij76] originally proposed that a similar constraint be imposed on statements. However, the refinement calculus work removed the continuity constraint as it forces nondeterminism to be bounded (or finite) [Heh79]. Although of little practical importance, it is theoretically more elegant for an object-oriented refinement calculus to support unbounded nondeterminism (and hence to drop the continuity constraint). Unbounded nondeterminism allows such specifications as

\[
a : [a \in \mathbb{Z}]
\]

whereas a bounded nondeterministic specification can only use a finite set of integers:

\[
a : [a \in \text{Bound}..\text{Bound}]
\]

where \( \text{Bound} \) is an arbitrary integer.

### 5.4 Summary

This chapter has analysed the effects of the use of various type models for representing objects. A model has been proposed that supports embedded methods and the addition of attributes to subclasses. While this model has limitations, namely the lack of support
for unbounded nondeterminism, it must be emphasised that these constraints should not
be considered to be those of the object-oriented refinement calculus developed by this
thesis. The specification of Desideratum 5.2 allows subsequent chapters to abstract from
the details of the object model. This ensures that the currently proposed model can be
easily replaced by another model that does not suffer the stated limitations.

The main result of the chapter is Theorem 5.4. When the typing properties stated
by this theorem are combined with judicious type checks, the proposed object model
is shown to uphold Desideratum 5.2. In other words, the model successfully supports
subsumption where subclasses can have more attributes and methods are embedded.
Chapter 6

An Object-Oriented Language for Refinement

This chapter presents a wide-spectrum object-oriented language. It is termed ‘wide-spectrum’ as it supports specification statements and consequently enables the specification of object-oriented programs. Chapter 7 uses this language as the basis for an object-oriented refinement calculus. Various equivalence properties are provided which are intended to be used as ‘refinement rules’ in addition to those presented in Chapter 7.

The object-oriented language consists of the definition of constructs that are used to access or manipulate objects (§6.1). These constructs include field selection, field update and a new object specification statement. The object specification statement is a derivative of the specification statement that provides a shorthand for automatically constraining the attributes not listed in the specification’s frame. The definition of the object specification statement is vital for a clear, concise specification of an object.

The defined constructs can be used within a value semantics. To address practical concerns, Section 6.2 revisits the definitions in the context of a semantics for references. Both semantics are presented for two reasons. The semantics for values is important in its own right. Additionally, the definitions for the semantics for values act as a stepping stone to the understanding of the semantics for references. The semantics do not contradict and in fact coexist.

Having developed an appropriate object model in the previous chapter, an object syntax is now provided to abstract from the model’s details. This abstraction also allows the substitution of the object representation with other models that possess the subtyping properties discussed in Chapter 5.

A predicate transformer object type (referred to hereafter as an object type), with distinct fields \( f_1\ldots f_p \) of the respective types \( F_1\ldots F_p \) and distinct methods \( m_1\ldots m_q \) has the
syntax:

\[
\text{Object} \\
\quad \text{field } f_i \in 1..p : F_i \\
\quad \text{method } m_i \in 1..q \\
\end
\]

The method types can be deduced and for conciseness are omitted.

A predicate transformer object (instance) has the following syntax if the fields \( f_i \in 1..p \) have the respective values \( f v_i \in 1..p \) and the methods \( m_i \in 1..q \) have the respective values \( m v_i \in 1..q \):

\[
\text{object} \\
\quad \text{field } f_i \in 1..p : F_i := f v_i \\
\quad \text{method } m_i \in 1..q = m v_i \\
\end
\]

The field types may be omitted when the object’s type is provided in the context.

To allow easy object variable introductions, the type of the object can be omitted when it is introduced as the type can be deduced from the object (refer to Definition 6.1).

**Definition 6.1 (Scoped Object Definition (Semantics for Values))**

\[
\begin{align*}
\left< \text{var } o \ ::= \ \text{Object} \\
\quad \text{field } f_i \in 1..p : F_i := f v_i \\
\quad \text{method } m_i \in 1..q = m v_i \\
\end{align*}
\]

\[
\begin{align*}
\left< \text{Prog} \\
\end{align*}
\]

\[
\begin{align*}
\left< \text{var } o : \text{Object} \\
\quad \text{field } f_i \in 1..p : F_i \\
\quad \text{method } m_i \in 1..q \\
\end{align*}
\]

\[
\begin{align*}
\left< \text{object} \\
\quad \text{field } f_i \in 1..p : F_i := f v_i \\
\quad \text{method } m_i \in 1..q = m v_i \\
\end{align*}
\]

\[
\begin{align*}
\left< \text{Prog} \\
\end{align*}
\]

\[
\diamond
\]

### 6.1 Client Constructs

Given a syntax for objects and their types, predicate transformer semantics are now provided for the language constructs used to access and manipulate objects within a semantics for values. These constructs, termed *client constructs*, include field selection, internal
field update, object specification and method call. Distinctions are made between internal and external field updates and method calls. Internal constructs are those which exclusively update their host while external are those that exclusively update objects other than their host.

### 6.1.1 Field Selection

Field selection is used to extract the value of an object’s field. It is defined for both object variables and explicit object constructions. Field selection is performed as part of an expression, typically in an expression that may occur on the right hand side of an assignment, or guard of an iteration or alternation statement.

**Definition 6.2 (Object Field Selection)** Selecting a field is defined as object calculus field selection.

\[ o.f \equiv o \odot f \]

This new syntax for object field selection is intended to help provide an abstraction of objects. The provision of such an abstraction will aid the substitution of the current object model by an alternative model.

**Example 6.3 (Object Field Selection Example)** For example, given an integer \( p \) and an object \( o \) with an integer field \( f \) then the predicate transformer \( p := o.f \) is the predicate transformer \( p := o \odot f \).

Internal field selection is accomplished using the variable `self` in lieu of `o`, e.g., `self.f` selects the host’s `f` field.

### 6.1.2 Object Specifications (Semantics for Values)

Specification statements, as presented in Section 2.2 and Definition B.47, deal with the alteration of program variables (the frame). Consequently, given an object variable \( o \) with a Boolean field \( l \), the specification \( o:: [o.l = \text{true}] \) specifies the desire for the \( l \) field of \( o \) to be equated to \text{true}. However, it does not restrict any other attributes of \( o \). Typically it is desired that the other attributes be constrained to their original values. An object specification is a new construct that provides a shorthand for constraining the attributes not listed in the frame. For example, the object specification:

\[ p.f:: [\text{post}] \]

allows only the attribute \( f \) of object \( p \) to be modified. The term `attribute path` is used to denote a path to an attribute (or object), e.g., \( p.f \) and \( o \) are attribute paths. The modification of \( p \)’s attribute \( f \) is equivalent to the modification of \( p \) (needed to allow \( p.f \) to alter) and
CHAPTER 6. AN OBJECT-ORIENTED LANGUAGE FOR REFINEMENT

all attributes of \( p.f \) etc., but not all attributes of \( p \). All enclosed attributes in Figure 6.1 are eligible for modification.

Object specifications are defined using a conservative extension of classical specifications by adding constraints about which attributes can be modified—these are determined by the attribute paths in the frame of the object specification. An attribute path is treated as a sequence of identifiers. For example, the attribute path \( p.f \) is used as an abbreviation of the sequence \( \langle p, f \rangle \).

The only program variable that needs to be modified is \( p \). If a classical specification were used instead of an object specification, it would be of the form:

\[
p: [\text{post} \land \text{constraint}]\]

where constraint ensures that those attributes not in the attribute paths remain stable. The frame consists of the heads of the attribute path sequences.

From Figure 6.1 it can be seen that the set of attributes that may be modified for a particular attribute path is the union of all attributes leading to the attribute path and all attributes ‘under’ the attribute path. When attribute paths are treated as sequences, these are the prefix and postfix closures, respectively. Thus the calculation of the modifiable attributes is split into the calculation of the prefix and postfix closures.

The postfix closure of an attribute path \( s \) is denoted by \( \text{poc}(s) \). All attribute paths in the tree beneath \( s \) are included. Hence \( \text{poc}(s) \) returns a (finite) set of sequences of labels. Using the example from Figure 6.1, \( \text{poc}(p.f) = \{ p.f, p.f.a, p.f.b \} \). The postfix closure of \( s \) is the set of attributes of \( s \) and the postfix closure of those attributes.

\[
\text{poc}(s) \triangleq \{ s \} \cup \bigcup \{ i \in \text{attributes}(s) \bullet \text{poc}(s \setminus \{ i \}) \}
\]

The postfix closure is formed by including all attributes of \( s \), and then recursively including the attributes of those.

**Definition 6.4 (Attributes Function)** The attributes function is defined using the type of the object. Given \( s : \_ \text{Obj } F \),

\[
\text{attributes}(s) \triangleq \text{dom}(F)
\]
The prefix closure of the attribute path \( s \), denoted by \( \text{prec}(s) \), are those attribute paths that lead to \( s \). It is a set of sequences, for example, \( \text{prec}(p.f) = \{p, p.f\} \). The prefix closure of \( s \) is the combination of all prefixes of \( s \):

\[
\text{prec}(s) \equiv \{ j \in 1..\#s \mid s \uparrow j \}
\]

where \( s \uparrow j \) denotes the prefix of \( s \) of length \( j \).

**Definition 6.5 (Object Specification (Semantics for Values))** The frame of the equivalent classical specification is the heads of each attribute path in the frame, i.e., \( \text{head}\{L\} \). These are termed the *bases*. The attributes in the prefix and postfix closures of the frame may be altered. The expression \( \text{frame} \) returns the prefix and postfix closures of the frame.

\[
\text{frame}(L) \equiv \bigcup\{s \in L \bullet \text{poc}(s) \cup \text{prec}(s)\}
\]

Those attributes not in the prefix and postfix closures must remain invariant. The expression \( \text{nonframe} \) returns those attributes not in the closure.

\[
\text{nonframe}(L) \equiv \bigcup\{s \in \text{head}\{L\} \bullet \text{poc}(s)\} \setminus \text{frame}(L)
\]

The constraint added to the classical specification’s postcondition is that all attributes, except those in the prefix and postfix closures of the frame, remain invariant\(^1\).

\[
L:: [\text{post}] \equiv \text{head}\{L\}; \bigwedge a \in \text{post} \land a \in \text{nonframe}(L) \bullet a = a_0
\]

\(^1\)This Definition is superseded later in the thesis by Definition Modified Object Specification (Semantics for Values) (7.14).
In this example, this is equivalent to:

\[ \forall a \in \{ p, p.f, p.f.a, p.f.b, p.g \}\setminus \{ p.f, p.f.a, p.f.b \} \cup \{ p, p.f \} \bullet a = a_0 \]

\[ \forall a \in \{ p.g \} \bullet a = a_0 \]

This means \( p.g \) must remain stable. The equivalent classical specification is therefore:

\[ p.f:: [post] \equiv p; [post \land p.g = p_0.g] \]

The overlap of closures in the frames of object specifications provides opportunities for modifying the object specification’s frame. An example of such frame modification is provided by Theorem 6.6.

**Theorem 6.6 (Expanding the Frame with a Postfix Closure)** Proof on page 205

The object specification frame may be extended with an attribute path \( p \) that is in the postfix closure of an attribute path \( o \) that is already in the frame.

\[ o:: [post] \equiv o, p:: [post] \]

provided \( p \in poc(o) \).

For the previous example,

\[ p.f:: [post] \equiv p.f.p.f.a:: [post] \]

as \( p.f.a \) is in the postfix closure of \( p.f \).

For convenience, the two predicate version of the object specification statement is also defined.

**Definition 6.7 (Dual Predicate Object Specification)**

\[ L:: [pre , post] \equiv \{ pre \} ; L:: [post] \]

\[ \diamond \]

**6.1.3 Field Update**

Object field updates are used to change the value of object fields. An update of an object \( o \) is defined as an assignment to \( o \) of an (object calculus) updated object.

**Definition 6.8 (Object Field Update (Semantics for Values))** Updating field \( f \) of object \( o \) with value \( e \) is defined as:

\[ o.f := e \equiv o := (o \triangleleft f \Leftarrow e) \]

The updated object is formed using object calculus method update to replace the body of field \( f \) with \( e \).
Object field updates can be introduced from object specifications as shown in Theorem 6.9.

Theorem 6.9 (Introduce Field Update (Semantics for Values) A) Proof on page 207
Given object value variable \( o \) and an expression \( e \) which may not include initial variables:

\[
o.f ::= [o.f = e[o\backslash o_0]] \equiv o.f := e
\]

6.1.4 Method Invocation

The methods of objects in our semantics are modelled as fields (with a predicate transformer type). When a method is called on an object, the field corresponding to the method name is selected from the object. The result of the field selection is then ‘executed.’

Definition 6.10 (Internal Object Method Invocation (Semantics for Values)) Invoking a method of the host reduces to object calculus field selection. Informally:

\[
call m \triangleq \text{self} \cdot m
\]

The value of the variable \( \text{self} \) is evaluated in the precondition state \( s \). For postcondition \( p \), internal method call is defined as:

\[
(call m) \, p \, s \triangleq s(\text{self}) \cdot m \, p \, s
\]

External Object Method Invocation

External object method invocation involves calling a method of an object that is not the host. This requires modifying the \( \text{self} \) variable to refer to the new object, an internal invocation, and finally, resetting \( \text{self} \) back to its original value. While \( \text{self} \) is set to the new ‘called’ object, the old value of \( \text{self} \) must be stored. The overall effect is to provide a stack of \( \text{self} \)’s. As a consequence of modelling states as records (Definition 4.16), the state type is not sophisticated enough to handle stacking. A similar effect can be achieved by introducing a fresh variable \( (v) \) to temporarily hold \( \text{self} \). The variable \( \text{self} \) is updated with the new host object \( (o) \) on which the method is called and an internal method call is made. After the method call, the current \( \text{self} \) is copied back to \( o \). The value of \( \text{self} \) is then reset to that temporarily held in \( v \). The type of \( \text{self} \) is switched between the original and new types by exiting one state space and entering the other. The statement \text{exit self}, as given in Definition B.35, is used to remove \( \text{self} \) from the state space. It is analogous to the closing brackets of a local variable block. The statement \text{enter self} : \text{Self} := v, as given in Definition B.33, is used to add \( \text{self} \) to the state space. It is analogous to the opening brackets of a local variable block.
Definition 6.11 (External Object Method Invocation (Semantics for Values))  For an object \( o : O \) and fresh variable \( v \),

\[
o . \text{call} \ m \triangleq \\
\left[\begin{array}{l}
\text{var} \ v : \text{Self} := self; \\
\text{exit} \ self;
\end{array}\right] \\
\left[\begin{array}{l}
\text{var} \ self : O \\
\text{self} := o; \ \text{call} \ m; \ o := self
\end{array}\right]
\]

This definition does not handle recursion. For instance, if the call to \( m \) updates the self object through \( o \), rather than \( self \), then the update to \( o \) after the call (\( o := self \)) would overwrite the alterations, reinstating \( o \)'s original value. One solution to this dilemma involves the use of reference variables. By stacking references to \( self \), rather than stacking the \( self \) values, all recursive calls on the stack dereference the same value. This is the approach taken for the reference semantics introduced in the next section.

### 6.2 Semantics for References

Object identity is a defining aspect of object-oriented languages. Therefore object-oriented languages usually incorporate a semantics based on object references. To provide such a capability, the current semantics for values is extended with a store. New constructs are introduced that augment yet do not replace the existing constructs. A set of object references \( \text{Ref} \) is introduced. An object reference is a location in the store: given an object reference, an object’s value is determined by indexing the store at the given location. The store is provided as a function from object references to objects:

\[
\text{Ref} \rightarrow \Gamma
\]

where \( \Gamma \) is the type of the objects in the store. To allow the use of aliasing control techniques, such as those presented by Utting [Utt97] and Bancroft [Ban97], multiple stores are allowed. Stores are held within the state. A store function called \( \text{store} \), containing objects of type \( \Gamma \), would use the following state type:

\[
\{\text{store} : (\text{Ref} \rightarrow \Gamma)\}_s
\]

The store is assumed to be finite to ensure that new objects can always be added.

For convenience, it is assumed that a store \( \text{store}^T \) exists for each type \( T \).
6.2.1 Objects (Semantics for References)

This section introduces new constructs for use with the extended semantics that incorporates references. Specifically, object construction and assignment are provided with additional, reference semantics definitions.

Object Construction Object constructions in a semantics for references introduce a variable of type Ref, allocate space in a store and copy the object into the store at the allocated location. For convenience the syntax Ref \{ store \} is used as a type for references that are allocated space in store store. Thus the program:

\[
\begin{align*}
\text{\texttt{\textbf{var} } o : \text{Ref} \{ \text{store} \} := \text{object}} \\
\text{field } f : F := f v \\
\text{method } m = m v \\
\text{\texttt{end}} \;
\end{align*}
\]

where \( F \) is not a reference, has the following semantics:

\[
\begin{align*}
\text{\texttt{\textbf{var} } o : \text{Ref}} \\
\text{o: \{ (\forall i \bullet o \not\in \text{dom store}) \};} \\
\text{store}_j(o) := \text{object} \\
\text{field } f : F := f v \\
\text{method } m = m v \\
\text{\texttt{end}}; \\
\text{Prog} \\
\end{align*}
\]

where the demonic choice non-deterministically chooses a reference that has not already been allocated. The quantifier \( i \) is used to range over all existing stores.

Assignment to a particular domain element in a function is defined as replacing the entire function with one updated at that point with the new value.

Definition 6.12 (Accessed Function Assignment)

\[
x(n) := e \triangleq x := x \oplus \{ n \mapsto e \}
\]

Consequently:

\[
\begin{align*}
\text{\texttt{\textbf{store} } (o) := \text{object}} \\
\text{field } f : F := f v \\
\text{method } m = m v \\
\text{\texttt{end}} \equiv \\
\text{\texttt{\textbf{store} } := \text{store} \oplus \{ o \mapsto \text{object}} \\
\text{field } f : F := f v \\
\text{method } m = m v \\
\text{\texttt{end}} \}
\end{align*}
\]
Within the value semantics, an object can have a field that is itself an object. For a reference semantics, to represent an object with a field that is itself an object, the convention is taken that the field type is a reference. Consequently the construction of composite objects may take multiple steps and the ‘value’ of an object can only be determined in the context of the store (wherein all references can be dereferenced). A composite object is constructed by defining the nested objects, entering them into the store and using these references to form the overall object. For example, the code:

\[
\begin{align*}
\text{var} \; o & : \text{Ref} \; \{\text{store}\} := \text{object} \\
\text{field} \; f & : F := fv \\
\text{field} \; \text{ref} & : \text{Ref} \; \{\text{store}_k\} := \text{object} \\
& \text{field} \; y : Y := yv \\
& \text{method} \; m = mv \\
\text{method} \; m & = mv \\
\text{end} \; \text{\textbullet} \\
\text{Prog} & \\
\end{align*}
\]

has the semantics:

\[
\begin{align*}
\text{var} \; o & : \text{Ref} \; \text{\textbullet} \\
\text{var} \; \text{freevar} & : \text{Ref} \; \text{\textbullet} \\
\text{freevar} & : \left(\forall i \; \bullet \; \text{freevar} \notin \text{dom} \; \text{store}_i\right) ; \\
\text{store}_k(\text{freevar}) & := \text{object} \\
& \text{field} \; y : Y := yv \\
& \text{end} ; \\
o & : \left(\forall i \; \bullet \; o \notin \text{dom} \; \text{store}_i\right) ; \\
\text{store}_j(o) & := \text{object} \\
& \text{field} \; f : F := fv \\
& \text{field} \; \text{ref} : \text{Ref} := \text{freevar} \\
& \text{method} \; m = mv \\
& \text{end} \\
\text{Prog} & \\
\end{align*}
\]

### 6.2.2 Client Constructs (Semantics for References)

The client constructs introduced in Section 6.1 were defined for a value semantics. These constructs are revisited in this section in the context of a semantics for references. The main difference for each construct of the semantics for values and references is, of course, that the semantics for references construct must use the store in the evaluation of an object’s value.
CHAPTER 6. AN OBJECT-ORIENTED LANGUAGE FOR REFINEMENT

Object Dereference

To encourage good practice for programming with references, a pragmatic constraint is used to prevent the direct access or manipulation of stores. To obtain the value of an object stored in a reference variable, object dereference is used. Object dereference returns the value of the object in the store.

Definition 6.13 (Object Dereference) Given a store A, the dereference of object o is defined as follows.

\[ o_A \triangleq A(o) \]

The store subscript (A) can be omitted when obvious from the context. A zero subscripted shorthand is also defined to denote the initial value of an object dereference.

\[ o_0 \triangleq A_0(o) \]

Field Selection

Field selection requires no new definition although the reference must be dereferenced before selection can occur.

Example 6.14 (Field Selection using References) Given a reference object o with a field f the following is used to access f.

\[ o^f \]

That is, the value of o is obtained by dereference, whereupon Definition 6.2 is used to obtain the value of f.

\[ o^f \equiv \text{store}(o)_f \]

Object Specifications (Semantics for References)

Similar constraints to those required for object specifications in a semantics for values are also needed for object specifications in a semantics for references.

Example 6.15 (Object Specifications Constraints) The program state in Figure 6.2 has three reference variables, o, p, and q and one integer variable r. Object o has a field f which is aliased to reference p, that is \( o.f = p \). Object p has a field h which is aliased to q. Object p also has a boolean field g. Object q has an integer field i.

Reference Attribute Paths: To alter the reference f in the object o, the following reference attribute path syntax should be used:

\[ o^f \]
Dereference Attribute Paths: The dereference attribute path

\[ o \uparrow f \uparrow \]

is used to denote that any attributes of the object \( o \uparrow f \uparrow \) may be modified. That is, the attributes \( g \) and \( h \) may be modified.

Reference Closure: The following attribute path form is reference closure:

\[ o \uparrow f ! \]

Reference closure is used when all attributes of all objects reachable by references through \( o \uparrow f \) may be altered. That is, the attributes of \( p \uparrow \) and \( q \uparrow \), but not \( o \uparrow f \), may be modified.

To simplify the definition of object specifications, the frame is split into four differing types, \( B \) for base attribute paths (e.g., \( o \) or \( p \)), \( D \) for dereference attribute paths (e.g., \( o \uparrow f \uparrow \)), \( R \) for reference attribute paths (e.g., \( o \uparrow f \)) and \( C \) for reference closure attribute paths (e.g., \( o \uparrow f ! \)). Also, a function \( \zeta(D,R,C) : \text{Ref} \rightarrow \mathbb{P} \text{Attributes} \) which maps references to the attributes that may be modified at that reference is used. The formation of \( \zeta \) is decomposed into three parts—one for each of \( D, R \) and \( C \).

\( D \): For the dereference attribute path \( o \uparrow f \uparrow \):

\[ \zeta = \{ o \uparrow f \mapsto \{g,h\}\} \]
indicating that any attribute of the object in store location \( o \uparrow f \) (or equivalently, via aliasing, \( p \)) may be modified. In general, for dereference attribute paths, \( \zeta \) is defined as:

\[
\{ i \in D \mid i \mapsto \text{attributes}(i) \}
\]

**R:** For the reference attribute path \( o \uparrow f \):

\[
\zeta = \{ o \mapsto \{ f \} \}
\]

indicating that the store can only be altered at the index \( o \), and even then, only attribute \( f \) of \( o \) may be altered. Since attribute paths are treated as sequences, the general definition of \( \zeta \) for reference attribute paths is:

\[
\{ i \in R \mid \text{front}(i) \mapsto \text{last}(i) \}
\]

where \( \text{front}(i) \) returns all but the last element of the sequence \( i \) and \( \text{last}(i) \) returns the element at the end of the sequence. Consequently, for the reference attribute path \( o \uparrow f \cdot h \) the set \( \zeta \) is:

\[
\zeta = \{ o \uparrow f \mapsto \{ h \} \}
\]

**C:** For the reference closure attribute path \( o \uparrow f! \):

\[
\zeta = \{ o \uparrow f \mapsto \{ g, h \}, o \uparrow f \cdot h \mapsto \{ i \} \}
\]

Aliasing can be used to produce the following, more readable, equivalent set.

\[
\zeta = \{ p \mapsto \{ g, h \}, q \mapsto \{ i \} \}
\]

The value of \( \zeta \) for a reference closure attribute path is calculated recursively. For a particular reference closure attribute path \( c \in C \), \( \zeta \) includes the mapping from \( c \) to its attributes and then recursively includes the reference closure of each attribute that is a reference. The function \( \text{refcl}(c) : \text{Ref} \rightarrow \mathcal{P}\text{Attributes} \) is used to calculate reference closures:

\[
\text{refcl}(c) \triangleq \left( \mu X \cdot \{ c \mapsto \text{attributes}(c) \} \cup \left( \begin{array}{c}
\text{ref} \mapsto \text{attrs} : X \cdot \\
\{ a \in \text{attrs} \mid \tau(a) = \text{Ref} \cdot \\
\text{ref} \cdot a \mapsto \text{attributes}(\text{ref} \cdot a) \}
\end{array} \right) \right)
\]

The condition \( \tau(a) = \text{Ref} \) is used to select only reference attributes. A least fixed-point definition is provided to account for reference loops, e.g., an object with a reference to itself. Using the Knaster-Taski theorem [BvW98, Theorem 19.1], instantiated with the subset-equality ordering and knowledge of the monotonicity of set union, the least fixed-point can be shown to exist.

To calculate \( \zeta \) over \( C \) the image of \( \text{refcl} \) is taken.

\[
\text{refcl}([ C ])\]
The definition of $\zeta$ is achieved by unioning the individual attribute path cases:

$$\zeta(D, R, C) \triangleq \{i \in D \bullet i \mapsto \text{attributes}(i)\} \cup \{i \in R \bullet \text{front}(i) \mapsto \text{last}(i)\} \cup \text{refcl}(\{C\})$$

where for each domain element the operator $\cup$ unions their ranges. For example,

$$\{m \mapsto \{p\}, n \mapsto \{q\}\} \cup \{m \mapsto \{r\}\} \equiv \{m \mapsto \{p, r\}, n \mapsto \{q\}\}$$

Thus $\cup$ is defined as:

$$j \cup k \triangleq \{i \in \text{dom}j \cap \text{dom}k \bullet i \mapsto j(i)\} \cup \{i \in \text{dom}k \bullet i \mapsto k(i)\} \cup \{i \in \text{dom}j \cap \text{dom}k \bullet i \mapsto j(i) \cup k(i)\}$$

The first set is for elements only in the domain of $j$, the second for elements only in the domain of $k$, the third for elements in both.

Now the constraints to be applied to the store can be determined. Given $\zeta$, the domain of $\zeta$ is the set of references that may be altered. One of the constraints to be added is that store should remain stable at all indices except those that may be altered. This set of indices is calculated by taking the domain of $\zeta$ from the domain of $\text{store}$:

$$\forall n \in \text{dom}(\text{store}) \setminus \text{dom}\zeta(D, R, C) \bullet \text{store}(n) = \text{store}_0(n)$$

For $\zeta(D, R, C) = \{p \mapsto \{g\}\}$ the attribute $h$ of object $p|\!|f$ should remain invariant. The following constraint ensures this. For each modifiable store position, the attributes not in the associated set of modifiable attributes (determined using $\zeta$), should remain constant.

$$\forall \text{ref} \in \text{dom}(\zeta(D, R, C)) \bullet \forall a \in \text{attributes}(\text{ref}) \setminus (\zeta(D, R, C)(\text{ref})) \bullet \text{ref}^{|a} = \text{ref}^{|a}_0$$

Collecting both of the above constraints provides the following definition.

**Definition 6.16 (Object Specification (Semantics for References))** Object specifications for a semantics of references, using the store function $\text{store}$, are defined as:

$$B, D, R, C:: [\text{post}]^\text{store} \triangleq \begin{aligned}
B, \text{store}:: & \text{post} \wedge \left[ \begin{array}{l}
\forall n \in \text{dom}(\text{store}) \setminus \text{dom}\zeta(D, R, C) \bullet \text{store}(n) = \text{store}_0(n) \\
\forall \text{ref} \in \text{dom}(\zeta(D, R, C)) \bullet \\
\forall a \in \text{attributes}(\text{ref}) \setminus (\zeta(D, R, C)(\text{ref})) \bullet \\
\text{ref}^{|a} = \text{ref}^{|a}_0
\end{array} \right]
\end{aligned}$$

where $B$ is the base attribute paths, (e.g., $o \circ p$), $D$ is the dereference attribute paths (e.g., $o \uparrow f \downarrow$), $R$ is the reference attribute paths (e.g., $o \uparrow f$) and $C$ is the reference closure attribute paths (e.g., $o \uparrow f!$), and $\zeta(D, R, C)$ is a function mapping store indices to the set of attributes which may be altered at that location. The superscript $\text{store}$ is omitted when obvious from the context\(^2\).

\(^2\)This Definition is superseded later in the thesis by Definition Modified Object Specification (Semantics for References) (7.15).
Theorem 6.17 (Extending the Frame with a Reference Closure) Proof on page 206
For object specifications using reference attribute paths, the frame can be extended with a
reference \((p)\) in the reference closure of a reference closure attribute path \((o!)\).
\[
o!:: [post] \equiv p, o!:: [post]
\]
provided \(\{\text{front}(p) \mapsto \text{last}(p)\} \in \text{refcl}(o)\).

Field Update

Definition 6.18 (Field Update (Semantics for References)) Like value semantics field
update, the reference semantics version reduces field update to object calculus method
update.
\[
o \uparrow f := e \equiv \text{store}(o) := \text{store}(o) \cup f \uplus e
\]

The following theorem can be used to replace an appropriate object specification with
a field update.

Theorem 6.19 (Introduce Field Update (Semantics for References) A) Proof on page
207 Assume an object reference \(o\) and expression \(e\). An object specification where the
attribute \(f\) of \(o\) may be altered such that its final value is \(e\) is equivalent to updating field
\(f\) of \(o\) with \(e\):
\[
o \uparrow f :: \{ o \uparrow f = e_0 \} \equiv o \uparrow f := e
\]

where \(e_0\) is \(e\) with all dereferences of \(\text{store}\) replaced by corresponding dereferences of
\(\text{store}_0\).

Internal Method Invocation (Semantics for References)

The definition of internal method invocation is similar to the approach taken within the
value semantics version—with the exception that the store is used to calculate \(\text{self}\). Informally:
\[
\text{self} \uparrow \text{store} \cdot \text{call} m \equiv \text{store}(\text{self}) \cup m
\]

Definition 6.20 (Internal Method Invocation (Semantics for References)) Formally,
for postcondition \(p\), the method \(m\) is invoked on the object \(\text{store}(\text{self})\) as determined by
the precondition state \(s\).
\[
\text{self} \uparrow \text{store} \cdot \text{call} m \ p s \equiv s(\text{store})(s(\text{self})) \cup m \ p s
\]
CHAPTER 6. AN OBJECT-ORIENTED LANGUAGE FOR REFINEMENT

External Method Invocation (Semantics for References)

The value semantics version of external method invocation used a technique of stacking object values. That technique did not support recursion. This problem is avoided by stacking object references instead of object values as any ‘aliased’ references created by recursive calls point to the same object value.

Definition 6.21 (External Method Invocation (Semantics for References))  For a fresh variable v,

\[
o \uparrow \text{call } m \triangleq \left[ \begin{array}{c}
\text{var } v : \text{Ref} \quad \bullet \ v, \ self := \ self, \ o; \quad \ self \uparrow \text{.call } m; \quad self := v
\end{array} \right]
\]

6.3 Summary

This chapter has defined a wide spectrum, object-oriented language which enables the specification of object-oriented programs. The language will be used as the basis for the object-oriented refinement calculus presented in Chapter 7. To enhance the reusability of the techniques and properties of the calculus, this language has been developed on an abstraction of objects. The abstraction dictates that objects must support attribute selection and update and subtyping properties as discussed in Chapter 5.

The language developed supports object construction, field selection, field update, method invocation, and object specifications for both a semantics for values and references. Object specifications have been shown to be an effective shorthand for automatically constraining attributes not listed in the frame of the specification. The definition of the object specification statement is vital for a clear, concise specification of an object.

Various equivalence properties of the constructs are provided. These properties are intended to be used as ‘refinement rules’ in addition to those presented in Chapter 7.
Chapter 7

Object- and Class-Refinement

Building on the object-based specification language of the previous chapter, an object-based refinement calculus is developed by exploring an object-refinement relation and providing related object-refinement rules. The calculus is then extended to a class-based refinement calculus.

The calculus has been designed to allow statements to be object-refinement monotonic. That is, it is a refinement to replace an object with an object-refinement. For example, given classes $C$ and $C'$ such that $C'$ is an object-refinement of $C$, that is, $C \sqsubseteq C'$, and client code $Prog$ then

$$
\left[ \text{var } x : \text{Ref} \bullet x := \text{new } C; \ Prog \right] 
\subseteq 
\left[ \text{var } x : \text{Ref} \bullet x := \text{new } C'; \ Prog \right]
$$

This property, termed object-refinement simulation, is similar to Theorem 1 from [BMvW00] and is generalised by Theorem 7.27 on page 91. The generalisation is termed construction monotonicity and considers the specific refinement of the new construct.

The object-refinement relations of other object-oriented refinement calculi are typically based on data refinement. That approach is motivated by the need to change an object’s type, e.g., when new attributes are introduced to form a subtype. This is aesthetically displeasing as data refinement need only be used when an object’s state space is altered, not merely extended. Since, by subsumption, the objects with additional attributes are also objects of the original type, the state space does not need to be altered. Hence data refinement is not required. For this reason, the classical data refinement approach is supplemented here with a novel object-refinement relation based on an algorithmic refinement approach. This relation, defined in Section 7.1, handles the object-refinement of an object to another of either the same type or a subtype. Section 7.2 illustrates the use of the algorithmic-refinement-based relation for the incremental refinement of objects and their clients. Corollary 7.28 formalises object-refinement simulation as a specific result of construction monotonicity.

Section 7.3 illustrates the use of subsumption for encapsulating an object’s attributes and hence provides a model for private attributes. This permits the private attributes of
an object to be data refined without modifying its clients. A more flexible (than object-refinement) data-refinement-based relation, introduced in Section 7.4, allows the refinement of an object to one that is not a subtype. The tradeoff, however, is that the entire object must be data refined as the data refinement of a statement is not in general monotonic. The relation is constrained such that the refined object’s type is a subtype of the object’s declaration type, i.e., the type of the variable holding the object. These techniques, relations and properties are extended to a class-based approach in Section 7.5.

7.1 Algorithmic Object-Refinement

This section introduces an object-refinement relation based on an algorithmic refinement approach and presents object-based refinement rules. For instance, rules are presented for object-refining attributes, and for adding new attributes.

Algorithmic object-refinement is defined so that it is possible to substitute (specification) objects with object-refinements—implementation objects. That is, object-refinement is defined to guarantee behavioural consistency. One way to ensure this is to force the methods of the implementation object to be refinements of those of the specification object. Additionally, for basic field types, e.g. integers, the fields need to be equal, and for object field types, the implementation object’s field needs to be an object-refinement of the corresponding specification object’s field.

An object \( \sigma \) is an object-refinement of another \( \theta \) if the following conditions hold:

- the fields of \( \sigma \) which are of basic types (e.g., Booleans) are equal to the respective fields of \( \theta \),
- the methods of \( \sigma \) are respectively refined by those in \( \theta \), and
- the fields of \( \sigma \) which are of object types are object-refined by the respective fields of \( \theta \).

Since our objects are finite, recursive object-refinements unfold to object-refinements of fields of basic types and methods.

**Definition 7.1 (Object-Refinement Algorithmic)** For objects \( \text{impl} : \text{Impl} \) and \( \text{spec} : \text{Spec} \), where \( \text{Impl} \) and \( \text{Spec} \) are basic types, object-refinement is equality:

\[
\text{spec} \sqsubseteq \text{impl} \iff \text{spec} = \text{impl}
\]

For objects \( \text{impl} : \text{Impl} \) and \( \text{spec} : \text{Spec} \), where \( \text{Impl} \) and \( \text{Spec} \) are function types such that \( \text{Impl} \preceq \text{Spec} \):

\[
\text{spec} \sqsubseteq \text{impl} \iff \forall j \in \text{dom spec} \bullet \text{spec}_{\circ j} \sqsubseteq \text{impl}_{\circ j}
\]
For objects $impl : Impl$ and $spec : Spec$, where $Impl$ and $Spec$ are object types such that $Impl \preceq Spec$, and where $spec$ has fields $\text{fields}(spec)$ and methods $\text{methods}(spec)$:

$$spec \sqsubseteq impl \equiv \forall j \in \text{dom}\ fields(spec) \bullet spec \circ j \sqsubseteq impl \circ j \land \forall j \in \text{dom}\ methods(spec) \bullet spec \circ j \sqsubseteq impl \circ j$$

Like classical refinement, for object-refinement to be useful in practice it must be a preorder. This relation is a preorder as it is reflexive (Theorem B.64), and transitive (Theorem B.65).

**Object-Refinement (Semantics for References)** In a semantics for values, the notion of object-refinement is applicable to nested objects: objects that have fields containing other objects. The object-refinement of a field results in the object-refinement of its host. Within a semantics for references, objects are, by convention, single-level entities as their fields are of basic types (including references). Consequently, the nested notion of object-refinement has little practical significance as object-refinement on basic types reduces to equality. To effect the refinement of an object’s field, the object at the referenced store location must be separately object-refined. In summary, whereas for a semantics of values, object-refinement has a recursive nature, for a semantics for references, object-refinement has a single, top-level nature. In practice, object-refinement of an object’s field for a semantics for references can be achieved, nontrivially, by refining the store.

The following refinement rules can be used to refine an object’s attributes and to add new attributes to an object.

**Theorem 7.2 (Update Object Field)** Proof on page 208 Replacing an object’s field with an object-refinement is an object-refinement. If object $o$ has field $l$ and $o \circ l \sqsubseteq f$, then $o$’s $l$ field can be replaced by $f$.

$$\frac{o \circ l \sqsubseteq f}{o \sqsubseteq (o \circ l \lll f)}$$

The construct $o \circ l \lll f$ (introduced in Section 4.1) is object calculus syntax representing the replacement of $o$’s field $l$ with $f$.

**Example 7.3 (Update Object Field)** For example, for a semantics for values, this rule
can be used to show that

```object
field incrementor ::= object
  field val : Z := 8
  method inc = val := val + 1 \& val := val + 2
end
end
```

provided, as shown in Example 7.6, that

```object
field val : Z := 8
method inc = val := val + 1 \& val := val + 2
end
end
```

Theorem 7.2 has the following corollary.

**Theorem 7.4 (Update Object Field Refined)** Proof on page 209 If \( e \) object-refines to \( f \), then updating any field of any object \( o \) with \( e \) object-refines to updating \( o \) with \( f \) instead.

\[-\]

\[ e \sqsubseteq^s f \]

\[ (o \circ fld \models e) \sqsubseteq^s (o \circ fld \models f) \]

**Theorem 7.5 (Update Object Method)** Proof on page 209 This refinement rule permits the refinement of an object’s method, resulting in an object-refinement. Given an object \( o \) with a method \( l \), and a method \( m \) that is a refinement of \( o \circ l \), then the replacement of \( o \circ l \) with \( m \) is a valid object-refinement.

\[-\]

\[ o \circ l \sqsubseteq m \]

\[ o \sqsubseteq^s (o \circ l \models m) \]
Example 7.6 (Update Object Method) This example completes Example 7.3 by proving its proviso. The following object-refinement is valid since $val := val + 1 \sqcap val := val + 2$ refines to $val := val + 1$.

```
object
  field val : Z := 8
  method inc = val := val + 1 \sqcap val := val + 2
end
```

The following theorem permits the addition of new attributes to an object.

**Theorem 7.7 (Introduce Object Attributes)** Proof on page 209 Given an object with fields $f_{1..i}$ and methods $m_{1..k}$, adding new fields $f_{i+1..i+j}$ (for $j \geq 0$) and methods $m_{k+1..k+p}$ (for $p \geq 0$) to an object produces an object-refinement. For field values $f_{v_{1..i+j}}$ of types $F_{1..i+j}$, and methods $m_{v_{1..k+p}}$:

```
object
  field ${h \in 1..j : F_h := f_{v_h}}$
  method $m_{h \in 1..k} = m_{v_h}$
end
```

7.2 Refinement of Client Constructs

This section presents several refinement rules involving client constructs: the language constructs that use objects. Without restrictions, object-refinement is not monotonic. Section 7.2.1 shows a solution, inspired by Naumann [Nau94b], that constrains the semantics, allowing object-refinement to be monotonic. Section 7.2.2 presents the definition of the language construct `new` found in practical object-oriented languages. It is defined as a language construct for a semantics for references and provides the capability to non-deterministically choose a store location and clone (copy) an object into that location. Section 7.2.3 redefines the field update language constructs, to permit monotonic object-refinements of fields. Section 7.2.4 provides rules that allow method calls to be introduced.
CHAPTER 7. OBJECT- AND CLASS-REFINEMENT

7.2.1 Object-Assignment

When postconditions are permitted to characterise arbitrary sets of states, object-refinement is not monotonic. For instance, given objects $e$ and $f$ where $f$ is an object-refinement of $e$, $e \sqsubseteq^c f$, and an assignment statement that assigns $e$ to $o$, then it is not a valid refinement to replace $e$ with $f$:

$$o := e \not\sqsubseteq o := f$$

The assignment $o := e$ establishes the postcondition $o = e$. In contrast, the assignment $o := f$ does not establish $o = e$ unless $e = f$. To obtain object-refinement monotonicity, an approach similar to that suggested by Naumann [Nau94b] is taken: predicates are restricted to those that are monotonic under object-refinement. This is achieved by only permitting predicates that are upwards closed under object-refinement. In general, for a partially ordered set $X$ (with the ordering $\leq$), a subset $\phi$ of $X$ is upwards closed (up-closed) if

$$\forall a, b : X \bullet a \leq b \Rightarrow (a \in \phi \Rightarrow b \in \phi)$$

This property is specialised for predicates as follows.

**Definition 7.8 (Object-Refinement Monotonic Predicate)** A predicate

$$p \sqsupseteq \lambda x : \text{Object } \bullet q(x)$$

is monotonic under object-refinement if, for objects $e$ and $f$ such that $e \sqsubseteq^c f$, $p(e)$ entails $p(f)$.

$$\forall e, f \bullet e \sqsubseteq^c f \Rightarrow (p(e) \Rightarrow p(f))$$

The syntax $\text{Object } \sqcup$ denotes the empty object type, i.e., the supertype of all object types.

A pragmatic constraint is used to apply this monotonicity constraint to predicates. The application of the monotonicity constraint does not alter the semantics. However, it does restrict the predicates and predicate transformers that can be used for objects. In practice this restriction means that the predicates must be monotonic in the program’s object variables and logical constants, but not necessarily in explicit object constructions. For variable $o$ and explicit object $e$, predicates such as $o \sqsupseteq^c e$ are monotonic in $o$ and are consequently allowed. Predicates such as $o = e$ and $o \sqsubseteq^c e$ are not permitted as they are not monotonic in $o$: they may be satisfied by the object $o$ but not by all proper object-refinements of $o$. 
For basic types the restriction to monotonic predicates has no significance. This is because object-refinement reduces to equality for these types and all predicates are monotonic under the equality relation: \( l = m \Rightarrow p(l) \equiv p(m) \). Consequently, the results of the classical refinement calculus are upheld. As such, this calculus is at least as expressive as the classical refinement calculus.

When specifying and subsequently refining a program, the specifier would choose upwards closed predicates. The monotonicity constraint forces a proof obligation to be generated for predicates involving variables or constants of an object type. This obligation will usually be trivially discharged. The presence of non-upwards closed predicates can often be determined syntactically, e.g., predicates equating objects or guards involving objects. Any legacy code from the classical refinement calculus is already monotonic. As such, “upwards closing” a non-upwards-closed predicate is a non-issue in the context of this thesis.

The language constructs must be shown to maintain upwards closure to ensure that the calculus does not introduce non-monotonic predicates.

**Definition 7.9 (Object-Refinement Monotonic Maintenance)** Given an object-refinement monotonic predicate \( p \), then the predicate transformer \( pt \) maintains upwards closure if \( pt(p) \) is also object-refinement monotonic.

Proofs that the standard guarded command language constructs maintain the monotonicity of predicates under refinement are presented by King [Kin99, p76]. One constraint imposed by King’s proofs is that the guards of alternation and iteration statements cannot involve objects as the same guard predicate occurs in both monotonic and anti-monotonic positions (refer to Definition Alternation Statement (B.56)). The proofs for object-refinement (instead of refinement) are essentially the same. King omits the proofs for logical constants and local variables, however, these are shown trivially given that the bodies of the logical constant and local variable blocks preserve object-refinement monotonicity.

To preclude specification statements from introducing non-monotonic predicates, King constrains the precondition to be monotonic and the postcondition to be anti-monotonic in variables. Theorem 7.10 generalises those results by relaxing the anti-monotonicity constraints in the postcondition on the variables in the frame. Consequently, only the variables in the postcondition that are not in the frame (and any initial variables) are required to be anti-monotonic.

A pragmatic constraint is used to apply these constraints to specification statements allowing them to be object-refinement monotonic. Similar to the application of the monotonicity constraint to predicates, the application of this constraint does not alter the semantics. However, it does alter the predicates that may be chosen for use within specification statements.
Theorem 7.10 (Specifications Object-Refinement Monotonicity)  Given

- program variables $\bar{x} \cup \bar{y}$ where $\bar{x}$ and $\bar{y}$ are disjoint;
- the object-refinement monotonicity of $pre(\bar{x}, \bar{y})$ for $\bar{x}$ and $\bar{y}$;
- the object-refinement anti-monotonicity of $post(\bar{x}, \bar{y}, y_0)$ for $\bar{x}$ and $y_0$;

then the specification

$$\bar{y}^* \colon [pre(\bar{x}, \bar{y}) \land post(\bar{x}, \bar{y}, y_0)]$$

preserves object-refinement monotonicity.

Proof

For any predicate $\phi(\bar{x}, \bar{y})$ that is object-refinement monotonic on the variables $\bar{x}$ and $\bar{y}$ and fresh variable vectors $\bar{x}'$ and $\bar{y}'$ such that $\bar{x} \subseteq \bar{x}'$ and $\bar{y} \subseteq \bar{y}'$, it must be shown that

$$\bar{y}^* \colon [pre(\bar{x}, \bar{y}) \land post(\bar{x}, \bar{y}, y_0)] \Rightarrow \phi(\bar{x}, \bar{y})$$

$\Leftrightarrow$ Specification Statement (B.47)

$$pre(\bar{x}, \bar{y}) \land (\forall \bar{y} \bullet post(\bar{x}, \bar{y}, y_0) \Rightarrow \phi(\bar{x}, \bar{y})) [y_0 \setminus \bar{y}]$$

$$\Rightarrow$$

$$pre(\bar{x}, \bar{y}) \land (\forall \bar{y} \bullet post(\bar{x}, \bar{y}, y_0) \Rightarrow \phi(\bar{x}, \bar{y})) [y_0 \setminus \bar{y}'][\bar{x}' \setminus \bar{x}, \bar{y}']$$

$\Leftrightarrow$ Substitution

$$pre(\bar{x}, \bar{y}) \land (\forall \bar{y} \bullet post(\bar{x}, \bar{y}, y_0) \Rightarrow \phi(\bar{x}, \bar{y})) [y_0 \setminus \bar{y}]$$

$$\Rightarrow$$

$$pre(\bar{x}', \bar{y}') \land (\forall \bar{y} \bullet post(\bar{x}, \bar{y}, y_0) \Rightarrow \phi(\bar{x}, \bar{y})) [y_0 \setminus \bar{y}'][\bar{x}' \setminus \bar{x}, \bar{y}']$$

$\Leftrightarrow$ Substitutions and renaming of bound quantifier

$$pre(\bar{x}, \bar{y}) \land (\forall \bar{z} \bullet post(\bar{x}, \bar{z}, \bar{y}) \Rightarrow \phi(\bar{x}, \bar{z}))$$

$$\Rightarrow$$

$$pre(\bar{x}', \bar{y}') \land (\forall \bar{z} \bullet post(\bar{x}, \bar{z}, \bar{y}) \Rightarrow \phi(\bar{x}, \bar{z}))$$

$\Leftrightarrow$ Monotonicity of $pre(\bar{x}, \bar{y})$ with respect to $\bar{x}$ and $\bar{y}$, $\bar{x} \subseteq \bar{x}'$ and $\bar{y} \subseteq \bar{y}'$.$$

$$(\forall \bar{z} \bullet post(\bar{x}, \bar{z}, \bar{y}) \Rightarrow \phi(\bar{x}, \bar{z}))$$

$$\Rightarrow$$

$$(\forall \bar{z} \bullet post(\bar{x}', \bar{z}, \bar{y}) \Rightarrow \phi(\bar{x}', \bar{z}))$$

$\Leftrightarrow$ Universal Quantification Weak Distribution (B.10)

$$(post(\bar{x}, \bar{z}, \bar{y}) \Rightarrow \phi(\bar{x}, \bar{z})) \Rightarrow (post(\bar{x}', \bar{z}, \bar{y}) \Rightarrow \phi(\bar{x}', \bar{z}))$$
Given that \( \text{post}(\bar{x}, \bar{y}, \bar{y}_0) \) is anti-monotonic for \( \bar{x} \) and \( \bar{y}_0 \), then \( \text{post}(\bar{x}', \bar{z}, \bar{y}') \) is anti-monotonic for \( \bar{x}' \) and \( \bar{y}' \).

\[
\iff \phi(\bar{x}, \bar{z}) \Rightarrow \phi(\bar{x}', \bar{z})
\]

This is shown using the assumption that \( \phi(\bar{x}, \bar{y}) \) is monotonic for \( \bar{x} \). Informally, the knowledge that \( \phi(\bar{x}, \bar{y}) \) is monotonic for \( \bar{y} \) is not required as the specification statement modifies \( \bar{y} \).

\[QED\]

By removing the constraint that frame variables need to be anti-monotonic in the post-condition, specifications such as (for variables \( o \) and \( p \))

\[
o: [ p \subseteq s o ]
\]

can be used.

The following refinement rules extend (or explicitly prove) the work of Naumann and King with the properties required in this thesis. The first two refinement rules show that object-refining objects in assignments and specification statements produces a refinement.

**Theorem 7.11 (Object-Refine in Assignment (Semantics for Values))**  
\[\text{Proof on page 211}\]

For variable \( o \) and expressions \( e \) and \( f \):

\[
e \subseteq s f \\
\frac{}{(o := e) \subseteq (o := f)}
\]

That is, substituting \( e \) for \( f \) in the assignment \( o := e \) produces the refined statement \( o := f \).

**Theorem 7.12 (Object-Refine in Specification (Semantics for Values))**  
\[\text{Proof on page 211}\]

For variable \( o \) and expressions \( e \) and \( f \):

\[
e \subseteq s f \\
\frac{}{o: [ e \subseteq s o ] \subseteq o: [ f \subseteq s o ]}
\]

The following theorem is the analogy of the simple specification refinement rule [Mor94].

**Theorem 7.13 (Object-Refinement Specification)**  
\[\text{Proof on page 211}\]

For an expression \( e \) that does not contain any initial variables:

\[
o: [ e[o \backslash o_0] \subseteq s o ] \equiv o := e
\]
So far in this section it has been established that the standard guarded command language constructs and the specification statement (with some additional constraints) preserve (object-refinement) monotonic predicates. All client constructs, excluding object specifications, also maintain object-refinement monotonicity as they are definitional extensions of language constructs that maintain object-refinement monotonicity. Object specifications require minor modifications to adhere to the additional constraints imposed upon specification statements. The modified definitions are as follows. The only change for the semantics for values definition is that the equivalence relation between the initial and final attribute values has been replaced by the object-refinement relation.

**Definition 7.14 (Modified Object Specification (Semantics for Values))** Refer to Definition 6.5 for further explanation.

\[
L :: [\text{post}] \equiv \text{head}(L, []; \text{post} \land \forall a \in \text{nonframe}(L) \bullet a_0 \sqsubseteq^e a)
\]

Similar changes are required for the semantics for references version.

**Definition 7.15 (Modified Object Specification (Semantics for References))** Refer to Definition 6.16 for further explanation.

\[
B, D, R, C :: [\text{post}] \equiv
\begin{align*}
\forall n \in \text{dom}(\text{store}) \setminus \text{dom}(\zeta(D, R, C)) \bullet store_0(n) \sqsubseteq^e \text{store}(n) \\
\forall a \in \text{attributes}(\text{refs}) \setminus (\zeta(D, R, C))(\text{refs}) \bullet \\
\text{refs}^\uparrow a.a \sqsubseteq^e \text{refs}^\uparrow.a
\end{align*}
\]

Under these modifications, Theorems Expanding the Frame with a Postfix Closure (6.6), Extending the Frame with a Reference Closure (6.17), Introduce Field Update (Semantics for Values) A (6.9) and Introduce Field Update (Semantics for References) A (6.19) must be generalised as they are only applicable when object-refinement reduces to equality, i.e., for non-object (or basic) types. The generalisations of Theorems 6.6 and 6.17 are provided by Theorems D.2 and D.4 respectively. Theorems 7.29 and 7.30 on page 92 are the appropriate generalisations of 6.9 and 6.19, respectively.

The following theorems allow the frames of object specifications to be expanded with new fields.

**Theorem 7.16 (Expand Frame New Fields (Semantics for Values))** Proof on page 212

In a semantics for values, given \(n \geq 0, j \geq 0, 0 \leq k \leq n, p \geq 0, r \geq j + 1\), frames \(\text{frame}_{i \in 1..j}\) that are each subsets of the fields \(f_{i \in 1..n}\):

\[
\forall i \in 1..j \bullet \text{frame}_i \subseteq \text{flds}
\]
where
\[ \text{flds} \triangleq \bigcup_{i=1}^{n} \{\text{self}.f_i\} \]

If it is also assumed that
\[
o \triangleq \text{object} \\
\text{field } f_{i \in 1..n} : F_i := f_{v_i} \\
\text{method } m_{i \in 1..j} = \text{frame}_{i}:: [q] \\
\text{method } m_{i \in 1..j+1..r} = m_{v_i} \\
\text{end} \]

and
\[
o' \triangleq \text{object} \\
\text{field } f_{i \in 1..n+p} : F_i := f_{v_i} \\
\text{method } m_{i \in 1..j} = \text{frame}_{i}, \text{self}.f_{n+1..n+p}:: [q] \\
\text{method } m_{i \in 1..j+1..r} = m_{v_i} \\
\text{end} \]

then \(o \sqsubseteq^c o'\).

Theorem 7.18 (Object Specification Weaken Precondition) \hspace{1cm} \text{Proof on page 214} \hspace{1cm} The precondition of an object specification can be weakened. Given \(\text{pre} \Rightarrow \text{pre}'\):

\[ L:: [\text{pre}, \text{post}] \sqsubseteq L:: [\text{pre}', \text{post}] \]
Theorem 7.19 (Object Specification Strengthen Postcondition)  
Proof on page 215
The postcondition of an object specification can be strengthened. Given
\[ \text{pre}[\text{head}(L) \land \text{post}] \Rightarrow \text{post} \]
then
\[ L:: [\text{pre} , \text{post}] \subseteq L:: [\text{pre} , \text{post}]' \]

Theorem 7.20 (Object Specification Contract Frame)  
Proof on page 215
The frame of an object specification can be contracted. Given an object with fields \( f \) and \( g \):
\[ o.f, o.g:: [\text{pre} , \text{post}] \subseteq o.f:: [\text{pre} , \text{post}] \]

Theorem 7.21 (Introduce Sequential Composition (Semantics for Values))  
Proof on page 216
An object specification can be refined to a sequential composition. For an object with fields \( f, g \) and \( h \):
\[ o.f, o.g:: [q] \]
\[ \begin{align*}
& \subseteq \\
& \left[ \left[ \text{con} O \bullet \\
& o.f:: [p] ; o.f, o.g:: [p[o_0 \setminus O] \land o.h \supseteq O.h \land o.g \supseteq O.g , q[o_0 \setminus O] \right] \\
\right] \right] \\
\end{align*} \]

Theorem 7.22 (Object Specification Alternation)  
Proof on page 217
An object specification can be refined to an alternation. Given \( \text{pre} \Rightarrow GG \) where \( GG \) is the disjunction of the guards \( G_{i \in 1..n} \):
\[ w:: [\text{pre} , \text{post}] \subseteq \text{if} \|_{i \in 1..n} G_i \rightarrow w:: [\text{pre} \land G_i , \text{post}] \]

To ensure that alternation statements maintain object-refinement monotonicity the guards cannot involve objects. Refer to page 84 for further discussion. Similarly, iteration statements, as introduced in the following rule, have the same constraint.

Theorem 7.23 (Object Specification Iteration)  
Proof on page 217
An object specification can be refined to an iteration. Given an object \( o \) with fields \( f \) and \( g \) (\( g \) is used only in the proof), the invariant \( inv \), the integer variant \( V \) and the disjunction of the guards \( GG \):
\[ o.f:: [inv , inv \land \neg GG] \]
\[ \subseteq \begin{align*}
& \text{do} (|_{i \in 1..n} G_i \rightarrow o.f:: [inv \land G_i , inv \land (0 \leq V < V_0)]) \\
\end{align*} \]
Neither \( inv \) or \( GG \) may contain initial variables.
For a semantics of references, Theorems 7.11 through 7.13 hold, but since $Ref$ is a basic type the theorems are of little practical significance as object-refinement reduces to equality. Object-assignment for a semantics for references is now discussed and analogies of the theorems presented above are provided.

An important construct for a semantics for references is the assignment of references (termed *reference assignment*). Reference assignment modifies an object reference to point to a different location in the store. It may break an alias that previously existed on the original store location and may also form a new alias on the new store location. The following syntax is used to denote the reference assignment of reference $p$ to reference $q$.

$$p := q$$

Given an object-refinement $s \uparrow$ of $q \downarrow$, it is not, in general, a refinement to substitute $s$ for $q$ as they may point to different store locations. For example, given $s$ and $q$ such that their dereferences are equal, yet they reference different store locations ($s \uparrow = q \downarrow \land s \neq q$) then the code

$$p := q; \quad q.f := e$$

has the effect of assigning $p$ to $q$ and then updating $q$’s $f$ field, and consequently also $p$’s $f$ field. Replacing $q$ with $s$ in the assignment would mean that the update of $p$’s field no longer occurs. This behaviour is not a refinement of the original.

Another form of assignment within a semantics for references is *dereference assignment*. Dereference assignment alters the value stored in a store location. This may cause, through aliasing, the alteration of other state variables. The following syntax is used to denote the dereference assignment of expression $p$ to the reference variable $o$.

$$o \downarrow := p$$

Since dereference assignments manipulate objects, opportunities arise for the object-refinement of the objects used. The following refinement rules are analogous to those provided earlier for a semantic for values.

**Theorem 7.24 (Object-Refine in Assignment (Semantics for References))**  
Proof on page 219  
Object-refining an object in a dereference assignment produces a refinement. For reference variable $o$ and expressions $e$ and $f$:

$$e \subseteq^c f$$

$$\frac{(o \downarrow := e) \subseteq (o \downarrow := f)}{ }$$

**Theorem 7.25 (Object-Refinement Specification (Semantics for References))**  
Proof on page 219  
This refinement is analogous to Simple Specification (B.52) except this rule is applicable to a semantics for references. Assume a reference variable $o$ and an
expression \( e \) which does not include initial variables. The alteration of the store at \( o \) to establish \( o' \) as an object-refinement of \( e \) can be implemented by assigning \( e \) to \( o' \):

\[
o':: \left[ e_0 \sqsubseteq^e o' \right] \sqsubseteq o' := e
\]

where \( e_0 \) is \( e \) with all dereferences of \( store \) replaced with corresponding dereferences of \( store_0 \).

### 7.2.2 Reference Cloning

Reference cloning is a construct designed specifically for a semantics for references to encapsulate the non-determinism of choosing a reference location; the refinement of which is a task for the compiler. Traditionally, the allocation of a reference location is accompanied by the initialisation of the data at that location. Reference cloning achieves these functions by copying an object value into an unused reference location. Given an object reference \( o \), the following assigns \( o \) to an unused store location and copies the object denoted by \( e \) into that store location.

\[
o := \text{new } e
\]

**Definition 7.26 (New Operator)** Given an expression \( e \) the **new** language construct is defined as follows.

\[
o := \text{new } store_j e \triangleq \left[ \begin{cases} \text{var } t : \text{Ref} \setminus t : \left[ \forall i \in t \notin \text{dom } store_i \right] ; \ t \uparrow_{store_j} := e ; \ o := t \end{cases} \right]
\]

The variable \( t \) is introduced as \( o \) may be free in \( e \). The subscript \( store_j \) may be omitted when obvious from the context.

Since multiple stores are used, \( t \) is chosen such that it is not in the domain of any store.

It is a refinement to object-refine \( e \) as **new** is defined using an object-assignment of \( e \) to \( o' \). This refinement rule is termed *construction monotonicity*.

**Theorem 7.27 (Construction Monotonicity)** Given objects \( e \) and \( f \),

\[
e \sqsubseteq^e f
\]

\[
o := \text{new } e \sqsubseteq o := \text{new } f
\]

**Proof**

The proof is a straightforward application of Theorem Object-Refine in Assignment (Semantics for References) (7.24).

*QED*

Using an implementation object to ‘simulate’ a specification object is often desirable as it may, for example, introduce efficiencies.
Corollary 7.28 (Simulation Object-Refinement) Given the specification object $Spec$, an object-refinement $Impl$, that is, $Spec \subseteq Impl$, and a store $store_{Spec}$, the program

$$\left[ \text{var } obj : \text{Ref} \{store_{Spec}\} \bullet obj := \text{new } Spec; \text{ Prog} \right]$$

may be refined to use $Impl$ instead:

$$\subseteq \left[ \text{var } obj : \text{Ref} \{store_{Spec}\} \bullet obj := \text{new } Impl; \text{ Prog} \right]$$

Notice that $obj$ is still in a store containing instances of type $\tau(\text{Spec})$. This is type correct as all instances of $\tau(\text{Impl})$ are by subsumption instances of $\tau(\text{Spec})$. As a consequence, however, $\text{Prog}$ is not permitted to use any of the attributes in $\tau(\text{Impl})$ that are not in $\tau(\text{Spec})$.

7.2.3 Field Updates

This section provides refinement rules for introducing field updates and for the object-refinement of the value being assigned in a field update construct.

Theorem 7.29 (Introduce Field Update (Semantics for Values)) Proof on page 221
A field update can be introduced to force a field $fld$ of object $o$ to be an object-refinement of an expression $e$ that does not include initial variables.

$$o.fld:: [ e[o\backslash o_0] \subseteq^e o.fld ] \subseteq o.fld := e$$

Theorem 7.30 (Introduce Field Update (Semantics for References)) Proof on page 221
For a semantics for references, a field update can be introduced to force a field $fld$ of object $o$ to be an object-refinement of an expression $e$ which does not include initial variables:

$$o\uparrow fld:: [ e_0 \subseteq^e o\uparrow fld ] \subseteq o\uparrow fld := e$$

where $e_0$ is $e$ with dereferences of $\text{store}$ replaced by corresponding dereferences of $store_0$.

---

1 The technique of construction monotonicity is revisited in Section 7.4. There, a more general technique, simulation, is introduced. Simulation provides possibilities for enhancement of the client code by allowing the original method calls to be replaced with (perhaps more efficient) calls to the new methods.
Theorem 7.31 (Object-Refine Field Update (Semantics for Values)) Proof on page 222
This refinement rule allows the expression being assigned in a field update to be object-refined.

\[
\begin{align*}
  e & \subseteq^* f \\
  (o.fld := e) & \subseteq (o.fld := f)
\end{align*}
\]

Theorem 7.32 (Object-Refine Field Update (Semantics for References)) Proof on page 222
This refinement rule allows the expression being assigned in a field update to be object-refined for a semantics for references.

\[
\begin{align*}
  e & \subseteq^* f \\
  (o\uparrow.fld := e) & \subseteq (o\uparrow.fld := f)
\end{align*}
\]

7.2.4 Introduce Method Calls

This section illustrates the refinement rules that can be used to introduce method calls.

Theorem 7.33 (Introduce Method Call (Semantics for Values)) Proof on page 222
For a semantics for values, given object \( o \):

\[
(o\circ m)[self\setminus o] \subseteq o.call m
\]

Theorem 7.34 (Introduce Method Call (Semantics for References)) Proof on page 223
Similarly, for a semantics for references, given an object \( o\uparrow \):

\[
(o\uparrow\circ m)[self\setminus o] \subseteq o\uparrow.call m
\]

Theorem 7.35 (Value Parameterised Method Call) Proof on page 224
This rule is essentially that of Morgan’s Law 11.2 [Mor94]. The rule permits the use of a value parameter in the method being introduced. Assume an actual value parameter \( a \), a formal value parameter \( f \) (both disjoint from \( w \)), and a method \( m \) of object \( o\):

\[
\text{method } m(\text{value } f : F)
\]

For a semantics for values, if

\[
w.f: [pre, post] \subseteq (o\circ m)[self\setminus o]
\]

with \( post \) containing no \( f \), then

\[
w.a: [pre[f\setminus a], post[f_0\setminus a_0]] \subseteq o.call m(a)
\]

Similarly, for a semantics for references, if

\[
w.f: [pre, post] \subseteq (o\uparrow\circ m)[self\setminus o]
\]

with \( post \) containing no \( f \), then

\[
w.a: [pre[f\setminus a], post[f_0\setminus a_0]] \subseteq o\uparrow.call m(a)
\]
Analogous rules exist for result (result) parameters (B.62) and value-result (value-result) parameters [Mor94].

The following example is loosely based on one from [Mik98] and similarly uses a value semantics.

**Example 7.36 (Improving a Bag)** An object Bag is defined to contain a bag of characters. Initially the bag is empty. Methods are provided to check if the bag is empty (Empty), add individual characters (Add), to nondeterministically choose a character (Choose), and to add another bag (AddAll). Given the bag union operation \( \cup \) and a program \( \text{Prog} \),

\[
\text{Bag} \triangleq \text{object} \\
\quad \text{field } b : \text{bag Char} := [] \\
\quad \text{method } \text{Empty} (\text{result } e : \mathbb{B}) \triangleq \\
\quad \quad e : [ e \Leftrightarrow (\text{self}.b = []) ] \\
\quad \text{method } \text{Add} (\text{value } c : \text{Char}) \triangleq \\
\quad \quad \text{self}.b :: [ \text{self}.b = \text{self}0.b \cup [c] ] \\
\quad \text{method } \text{Choose} (\text{result } c : \text{Char}) \triangleq \\
\quad \quad \text{self}.b, c :: [ \text{self}.b \neq [], \text{self}0.b = \text{self}.b \cup [c] ] \\
\quad \text{method } \text{AddAll} (\text{value } nb : \tau(\text{Bag})) \triangleq \\
\quad \quad \text{self}.b, nb :: [ \text{self}.b = \text{self}0.b \cup nb0.b ]
\]

where

\[
\tau(\text{Bag}) \triangleq \text{Object} \\
\quad \text{field } b : \text{bag Char} \\
\quad \text{method } \text{Empty} (\text{result } e : \mathbb{B}) \\
\quad \text{method } \text{Add} (\text{value } c : \text{Char}) \\
\quad \text{method } \text{Choose} (\text{result } c : \text{Char}) \\
\quad \text{method } \text{AddAll} (\text{value } nb : \tau(\text{Bag}))
\]

Bag could be considered an abstract object as it contains statements (i.e., the object specifications) that are not code.

This example refines Bag to both implement it and provide additional functionality. The object-refinement will be used to maintain a count of the elements in the bag. A new
field \( cnt \) is introduced to record the count. The bag count operator is \( \#^+ \).

**Bag**

- Expand Frame New Fields (Semantics for Values) (7.16)
- Object Specification Strengthen Postcondition (7.19)
- **object**
  - field \( b : \text{bag Char} := [] \)
  - field \( cnt : \mathbb{N} := 0 \)
  - method Empty(\textbf{result} \( e : \mathbb{B} \)) \( \cong e : [ e \leftrightarrow (self.b = []) ] \)
  - method Add(\textbf{value} \( c : \text{Char} \)) \( \cong 
    \begin{align*}
    \text{self.b, self.cnt} &:: \text{self.b} = \text{self}_0.b \sqcup [c] \\
    \text{self.cnt} &:: \text{self}_0.cnt + 1
    \end{align*} \)
  - method Choose(\textbf{result} \( c : \text{Char} \)) \( \cong 
    \begin{align*}
    \text{self.b, self.cnt} &:: \text{self.b} \neq [], \text{self}_0.b = \text{self.b} \sqcup [c] \\
    \text{self.cnt} &:: \text{self}_0.cnt - 1
    \end{align*} \)
  - method AddAll(\textbf{value} \( nb : (\text{Bag}) \)) \( \cong 
    \begin{align*}
    \text{self.b, nb, self.cnt} &:: \text{self.b} = \text{self}_0.b \sqcup nb_0.b \\
    \text{self.cnt} &:: \text{self}_0.cnt + \#^+ (nb_0.b)
    \end{align*} \)

end

Traditional refinement calculus techniques can be used to introduce an iteration within the \textit{AddAll} method. A local variable \textit{empty} will be used to determine whether the bag is empty or not. This variable is required as object variables are not permitted in the guard of an iteration statement\(^2\).

\[
\begin{align*}
\text{self.b, nb, self.cnt} &:: \text{self.b} = \text{self}_0.b \sqcup nb_0.b \\
\text{self.cnt} &:: \text{self}_0.cnt + \#^+ (nb_0.b)
\end{align*}
\]

\( \sqsupseteq \) Modified Object Specification (Semantics for Values) (7.14)

- Introduce Local Variable Block (B.55)
- Introduce Sequential Composition (B.40) - Introduce a logical constant \textit{EMPTY}.
- Remove Logical Constant (B.45) - \textit{EMPTY} is not used or needed.
- Fix Initial Value (B.46) - Introduce and initialise \( S \) and \( N \).

\[
\begin{align*}
\begin{cases}
\text{var empty : \mathbb{B} } \\
\text{empty : [ empty \leftrightarrow (nb.b = []) ] } \\
\text{con S, N } \\
\text{empty, self.b, nb, self.cnt} &:: \text{self} = S \wedge nb = N \wedge empty \leftrightarrow (nb.b = []) \\
\text{self.cnt} &:: \text{self}_0.b \sqcup nb_0.b \\
\text{self.cnt} &:: \text{self}_0.cnt + \#^+ (nb_0.b)
\end{cases}
\end{align*}
\]

The iteration, to be introduced, will progressively transfer all elements from the bag in \( nb \) into the bag of \( self \). The invariant is:

\[
\begin{align*}
\text{self.b} \sqcup nb.b = S.b \sqcup N.b \land self.cnt + \#^+ (nb.b) = S.cnt + \#^+ (N.b) \land \\
\text{empty} \leftrightarrow (nb.b = [])
\end{align*}
\]

\(^2\)Refer to page 84.
CHAPTER 7. OBJECT- AND CLASS-REFINEMENT

The conjunct \( \text{empty} \iff (nb.b = []) \) is included to detect the condition for terminating the iteration: when there are no elements left in the bag of \( nb \).

The predicates of the specification are modified to resemble the invariant (and the negation of the guard for the postcondition: \( \text{empty} \)). The postcondition is strengthened as follows:

\[
\begin{align*}
(sel\!f_0 = S \land nb_0 = N \land \text{empty}_0 \iff (nb_0.b = [])) \land \text{empty} \land \\
self.b \uplus nb.b = S.b \uplus N.b \land self.cnt + \#^+(nb.b) = S.cnt + \#^+(N.b) \land \\
\text{empty} \iff (nb.b = [])
\end{align*}
\]

\[\Rightarrow\] Substitutions

\[
\begin{align*}
\text{empty} \land self.b \uplus nb.b = self_0.b \uplus nb_0.b \land \\
self.cnt + \#^+(nb.b) = self_0.cnt + \#^+(nb_0.b) \land \text{empty} \iff (nb.b = [])
\end{align*}
\]

\[\Rightarrow\] Simplifying using empty

\[
\begin{align*}
self.b \uplus nb.b = self_0.b \uplus nb_0.b \land \\
self.cnt + \#^+(nb.b) = self_0.cnt + \#^+(nb_0.b) \land nb.b = []
\end{align*}
\]

\[\Rightarrow\] Simplifying using \( nb.b = [] \)

\[
\begin{align*}
self.b = self_0.b \uplus nb_0.b \land self.cnt = self_0.cnt + \#^+(nb_0.b)
\end{align*}
\]

The previous specification refines as follows:

\[\Box \text{ Object Specification Strengthen Postcondition (7.19)}\]

\[\Box \text{ Object Specification Weaken Precondition (7.18)}\]

\[\Box \text{ Result Parameterised Method Call (B.62)}\]

\[
\begin{align*}
| & \textbf{var} \ \text{empty} : \mathbb{B} & \bullet \\
& \text{nb.Empty}(\text{empty}); & \bullet \\
| & \textbf{con} \ S, N & \bullet \\
& \text{empty}, self.b, nb, self.cnt : & \\
& \text{self.b \uplus nb.b} = S.b \uplus N.b & \\
& self.cnt + \#^+(nb.b) = S.cnt + \#^+(N.b) & \\
& \text{empty} \iff (nb.b = []) & \\
\end{align*}
\]

\[
\begin{align*}
| & \text{empty \land self.b \uplus nb.b} = S.b \uplus N.b & \\
& self.cnt + \#^+(nb.b) = S.cnt + \#^+(N.b) & \\
& \text{empty} \iff (nb.b = []) & \\
\end{align*}
\]

\[\Box \text{ Focussing on the specification allows the following refinement:}\]

\[\Box \text{ Object Specification Iteration (7.23)}\]

\[
\begin{align*}
\textbf{do} & \mathbin{\leftarrow} \text{empty} \mathbin{\rightarrow} \\
& \text{empty, self.b, nb, self.cnt} : & \\
& \text{self.b \uplus nb.b} = S.b \uplus N.b & \\
& self.cnt + \#^+(nb.b) = S.cnt + \#^+(N.b) & \\
& \text{empty} \iff (nb.b = []) & \\
\end{align*}
\]

\[
\begin{align*}
& (0 \leq \#^+(nb.b) < \#^+(nb_0.b)) & \\
& \text{od}
\end{align*}
\]
A local variable \( c : \text{Char} \) is introduced to temporarily remember a bag element of \( nb.b \) while it is transferred to \( self.b \). Focussing on the specification allows the following refinement:

\[
\begin{align*}
\& \text{Modified Object Specification (Semantics for Values) (7.14)} \\
\& \text{Introduce Local Variable Block (B.55)} \\
\& \text{Introduce Sequential Composition (B.40)} \\
\end{align*}
\]

\[
\begin{align*}
\var c : \text{Char} \cdot \\
\con \text{EMPTY, } C, NB \cdot \\
\end{align*}
\]

\[
\begin{align*}
\begin{array}{l}
\text{empty, } c, nb : \\
\text{self.b} \uplus \text{nb.b} = \text{S.b} \uplus \text{N.b} \\
\text{self.cnt} + \#^+(\text{nb.b}) = \text{S.cnt} + \#^+(\text{N.b}) \\
\text{empty} \Leftrightarrow (\text{nb.b} = \mathbb{[]}) \land \neg \text{empty} \\
\text{self.b} \uplus \text{nb.b} \uplus [c] = \text{S.b} \uplus \text{N.b} \\
\text{self.cnt} + \#^+(\text{nb.b}) + 1 = \text{S.cnt} + \#^+(\text{N.b}) \\
\text{empty} \Leftrightarrow (\text{nb.b} = \mathbb{[]}) \\
(0 \leq \#^+(\text{nb.b}) < \#^+(\text{nb}._0, \text{b})) \\
\text{nb}._0, \text{b} = \text{nb} \uplus [c] \\
\end{array}
\end{align*}
\]

\[
\begin{align*}
\begin{array}{l}
\text{c, empty, } self.b, nb : \\
\text{self.b} \uplus \text{nb.b} = \text{S.b} \uplus \text{N.b} \\
\text{self.cnt} + \#^+(\text{nb.b}) = \text{S.cnt} + \#^+(\text{N.b}) \\
\text{empty} \Leftrightarrow (\text{nb.b} = \mathbb{[]}) \\
(0 \leq \#^+(\text{nb.b}) < \#^+(\text{NB.b})) \\
\text{NB.b} = \text{nb} \uplus [c] \\
\end{array}
\end{align*}
\]

The next refinement uses the following entailment to strengthen the postcondition of the first specification.

\[
\begin{align*}
\&(\text{self.b} \uplus \text{nb}._0, \text{b} = \text{S.b} \uplus \text{N.b} \land \text{self.cnt} + \#^+(\text{nb}._0, \text{b}) = \text{S.cnt} + \#^+(\text{N.b}) \land \\
\& \text{empty} \Leftrightarrow (\text{nb}._0, \text{b} = \mathbb{[]}) \land \neg \text{empty} \land \\
\& \text{nb}._0, \text{b} = \text{nb} \uplus [c] \land \text{nb.cnt} = \text{nb.cnt} - 1 \land \text{empty} \Leftrightarrow (\text{nb.b} = \mathbb{[]}) \\
\Rightarrow \text{Substitutions} \\
\&(\text{self.b} \uplus \text{nb.b} \uplus [c] = \text{S.b} \uplus \text{N.b} \land \\
\& \text{self.cnt} + \#^+(\text{nb.b} \uplus [c]) = \text{S.cnt} + \#^+(\text{N.b}) \land \\
\& \text{nb}._0, \text{b} \neq \mathbb{[]} \land \text{nb}._0, \text{b} = \text{nb} \uplus [c] \land \text{empty} \Leftrightarrow (\text{nb.b} = \mathbb{[]}) \\
\Rightarrow \text{self.b} \uplus \text{nb.b} \uplus [c] = \text{S.b} \uplus \text{N.b} \land \\
\& \text{self.cnt} + \#^+(\text{nb.b}) + 1 = \text{S.cnt} + \#^+(\text{N.b}) \land \\
\& \text{empty} \Leftrightarrow (\text{nb.b} = \mathbb{[]}) \land (0 \leq \#^+(\text{nb.b}) < \#^+(\text{nb}._0, \text{b}))
\end{align*}
\]
CHAPTER 7. OBJECT- AND CLASS-REFINEMENT

\[\square \text{Object Specification Contract Frame (7.20)}\]
\[\text{Remove } c, \text{ empty and } nb \text{ from second specification.}\]
\[\text{Strengthen Postcondition (B.54)}\]
\[\text{Weaken Precondition (B.53)}\]
\[\text{Object Specification Strengthen Postcondition (7.19)}\]
\[\text{To remove empty } \Leftrightarrow (nb.b = []) \land (0 \leq \#^+(nb.b) < \#^+(NB.b))\]
\[\text{and to substitute using the precondition.}\]
\[\text{Object Specification Weaken Precondition (7.18)}\]
\[\text{Remove Logical Constant (B.45)}\]

\[
\begin{align*}
\text{The first specification can be refined as follows.} \\
\text{empty, } c, nb: \quad & \begin{cases} 
nb_{0}.b = nb.b \uplus [c] \\
nb.b \neq [], \ nb_{0}.cnt = nb_{0}.cnt - 1 \\
\text{empty } \Leftrightarrow (nb.b = [])
\end{cases} \\
\text{self.b, self.cnt:} \quad & \begin{cases} 
self.b = self_{0}.b \uplus [c] \\
self.cnt = self_{0}.cnt + 1
\end{cases}
\end{align*}
\]

\[\square \text{Introduce Sequential Composition (B.40)}\]
\[\text{Contract Frame (B.51)}\]
\[\text{Strengthen Postcondition (B.54)}\]
\[\text{Weaken Precondition (B.53)}\]
\[\text{Remove Logical Constant (B.45)}\]
\[\text{c, nb: } \begin{cases} 
nb_{0}.b = nb.b \uplus [c] \\
nb.b \neq [], \ nb_{0}.b = nb.b \uplus [c] \land nb.cnt = nb.cnt - 1 \\
\text{empty: } \begin{cases} 
\text{empty } \Leftrightarrow (nb.b = [])
\end{cases}
\end{cases} \]

\[\square \text{Result Parameterised Method Call (B.62)}\]
\[\text{nb.call Choose(c); nb.call Empty(empty)}\]

The remaining specification can also be refined to a method call.

\[\begin{align*}
\text{self.b, self.cnt:} \quad & \begin{cases} 
self.b = self_{0}.b \uplus [c] \\
self.cnt = self_{0}.cnt + 1
\end{cases}
\end{align*}\]

\[\square \text{Value Parameterised Method Call (7.35)}\]
\[\text{self.call Add(c)}\]

Collecting all the code and removing the unused logical constants produces the following
refinement of the \textit{AddAll} method.

\begin{verbatim}
[ [ var empty : B ⇒
  nb.Empty(empty);
  do empty →
  [ [ var c : Char ⇒
    nb.call Choose(c); nb.call Empty(empty); self.call Add(c)
  ]]
  od ]]
\end{verbatim}

Since bag \textit{Char} is a basic type (and hence object-refinement reduces to equivalence), either Introduce Field Update (Semantics for Values) (7.29) or Introduce Field Update (Semantics for Values) A (6.9) can be used to refine \textit{Add} to field updates.

\begin{verbatim}
self.b, self.cnt: [ self.b = self0.b ⊕ [c] ]
set.cnt = set0.cnt + 1
\end{verbatim}

\begin{verbatim}
[ [ con SELF •
  self.cnt: [ self.cnt = self0.cnt + 1 ];
  self.b, self.cnt: [ self.cnt = SELF.cnt + 1 ∧ self.b = SELF.b ]
  self.b = SELF.b ⊕ [c];
  self.cnt = SELF.cnt + 1 ]]
\end{verbatim}

The first specification is refined to an update of \textit{cnt}.

\begin{verbatim}
self.cnt = self0.cnt + 1
\end{verbatim}

\begin{verbatim}
\end{verbatim}

The second specification is refined to an update to \textit{b}.

\begin{verbatim}
self.b = self0.b ⊕ [c]
\end{verbatim}
A sequential composition can be introduced into the Choose method in a similar manner.

\[
\text{self}.b, \text{c}, \text{self}.\text{cnt}:: \left[ \begin{array}{l}
\text{self}.b \neq [], \\
\text{self}0.b = \text{self}.b \uplus [c] \\
\text{self}.\text{cnt} = \text{self}0.\text{cnt} - 1
\end{array} \right]
\]

\[\begin{aligned}
&\text{Introduce Sequential Composition (Semantics for Values) (7.21)} \\
&\text{Object Specification Strengthen Postcondition (7.19)} \\
&\text{Object Specification Weaken Precondition (7.18)} \\
&\text{Object Specification Contract Frame (7.20)}
\end{aligned}\]

\[
\left[ \begin{array}{l}
\text{con SELF} \\
\text{self}.\text{cnt}:: [\text{self}.b \neq []] \\
\text{self}.\text{cnt} = \text{self}0.\text{cnt} - 1 \\
\text{self}.b, \text{c}, \text{self}.\text{cnt}:: \\
\left[ \begin{array}{l}
\text{self}.b \neq [] \\
\text{self}.b = \text{self}0.b \uplus [c] \\
\text{self}.\text{cnt} = \text{self}.\text{cnt} - 1 \\
\text{SELF}.b = \text{self}.b \uplus [c]
\end{array} \right]
\end{array} \right]
\]

The first specification is refined to an update of \text{cnt}.

\[
\text{self}.\text{cnt}:: [\text{self}.\text{cnt} = \text{self}0.\text{cnt} - 1]
\]

\[\begin{aligned}
&\text{Introduce Field Update (Semantics for Values) (7.29)} \\
&\text{self} := \text{self}.\text{cnt} - 1
\end{aligned}\]

The \text{Empty} method is refined to an assignment statement.

\[
e:: e \leftrightarrow (\text{self}.b = [])
\]

\[\begin{aligned}
&\text{Invariant} \\
e:: e \leftrightarrow (\text{self}.\text{cnt} = 0)
\end{aligned}\]

If an invariant were imposed on the object to constrain the number of elements in bag \text{b} to equal \text{cnt}, i.e., \#(\text{self}.b) = \text{self}.\text{cnt}, then the \text{Empty} method could be refined to use \text{cnt} instead:

\[
e:: e \leftrightarrow (\text{self}.b = [])
\]

\[\begin{aligned}
&\text{Invariant} \\
e:: e \leftrightarrow (\text{self}.\text{cnt} = 0)
\end{aligned}\]

When these refinements are applied to the object the following property is established

\text{Bag}

\[\begin{aligned}
&\text{Update Object Method (7.5)} \\
&\text{CountingBag}
\end{aligned}\]
CHAPTER 7. OBJECT- AND CLASS-REFINEMENT

where

\[
\begin{align*}
\text{CountingBag} & \triangleq \\
\text{object} & \\
\text{field} \ b : \ \text{bag} \ \text{Char} := [ ] \\
\text{field} \ cnt : \ \mathbb{N} := 0 \\
\text{method} \ \text{Empty}(\text{result} \ e : \ \text{B}) & \triangleq \\
\quad e := (\text{self}.b = [ ] ) \\
\text{method} \ \text{Add}(\text{value} \ c : \ \text{Char}) & \triangleq \\
\quad \text{self}.cnt := \text{self}.cnt + 1; \\
\quad \text{self}.b := \text{self}.b \uplus [ c ] \\
\text{method} \ \text{Choose}(\text{result} \ c : \ \text{Char}) & \triangleq \\
\quad \text{self}.cnt := \text{self}.cnt - 1; \\
\quad \text{self}.b, c:: [ \text{self}.b \neq [ ] , \ \text{self}_0.b = \text{self}.b \uplus [ c ] ] \\
\text{method} \ \text{AddAll}(\text{value} \ nb : \ \tau(\text{Bag})) & \triangleq \\
\quad [ \text{var} \ empty : \ \text{B} \bullet \\
\quad \text{nb}.\text{Empty}(\text{empty}); \\
\quad \text{do} \rightarrow \text{empty} \rightarrow \\
\quad \quad [ \text{var} \ c : \ \text{Char} \bullet \\
\quad \quad \text{nb}.\text{call} \ \text{Choose}(c); \ \text{nb}.\text{call} \ \text{Empty}(\text{empty}); \ \text{self}.\text{call} \ \text{Add}(c) \\
\quad \quad ] ] \\
\quad \text{od} \\
\text{end}
\end{align*}
\]

The Choose method remains a specification in this example. Later, Example 7.47 demonstrates one way this method could be refined to an implementation.

Given CountingBag, a program that uses the Bag object can be improved to use the CountingBag object instead. That is, assuming variable \( a : \tau(\text{Bag}) \),

\[
a := \text{Bag} \\
\triangleq \text{Object-Refine in Assignment (Semantics for Values) (7.11)} \\
a := \text{CountingBag}
\]

Similarly, given variable \( r : \text{Ref} \)

\[
r := \text{new Bag} \\
\triangleq \text{Construction Monotonicity (7.27)} \\
r := \text{new CountingBag}
\]

One difference between this development and Mikhajlova’s specification is that this version redefines the entire Add method in each subclass. Mikhajlova uses a ‘special identifier’ super, but the semantics are unclear. Since attributes are embedded in objects in this thesis, subclassing overrides attributes. Calls to super could not, therefore, be modelled by treating super as a special identifier which is updated by the definition of method invocation. One way to support super-calls would be to keep a copy of the methods of the superclass. Section 9.1 presents a more effective technique than the use of super-calls.
for the incremental extension of a class. One could also argue that the discontinuity that super-calls introduce outweigh their usefulness as code reuse mechanisms. They are used as the ‘goto’ of object-orientation: they are effective both at code reuse and reducing understanding.

Another difference between this development and Mikhajlova’s is the redeclaration of superclass fields. For unexplained reasons, Mikhajlova’s example specification redeclares and reinitialises (in the constructor) all fields of the superclass.

In the example above, reasoning about object specifications appears to rely upon the dynamic type environment. However, the definition and associated rules rely solely on the (static) most defined type operator. The most defined type operator, e.g., \( x : \bot C \), (defined on page 37) is used to constrain the static type. Care is needed to distinguish its effect in the calculus from that of the dynamic type-checking operator (\texttt{typecase}). The syntax \( x : \tau(Bag) \) can be used to define the static type environment rather than (repeatedly) subscripting specifications as follows:

\[
x.b::[x.b = \emptyset]_{\tau(Bag)}
\]

The object specification above has the (most defined) type

\[
\text{Ptrans} \{ x : \tau(Bag) \}_{\text{st}} \{ x : \tau(Bag) \}_{\text{st}}
\]

where \text{Ptrans} post pre. By subsumption, the object specification has other (static) types, e.g.,

\[
\text{Ptrans} \{ x : \tau(Bag) \}_{\text{st}} \{ x : \tau(CountingBag) \}_{\text{st}}
\]

as \( \tau(CountingBag) \) is a subtype of \( \tau(Bag) \). Informally, this allows the statement to accept a precondition where the variable \( x \) has the type \( \tau(CountingBag) \), but it will only pass back a \( \tau(Bag) \) object in the postcondition, effectively removing the \( cnt \) field from the returned object. A statement of type

\[
\text{Ptrans} \{ x : \tau(Bag) \}_{\text{st}} \{ x : \tau(Bag) \}_{\text{st}}
\]

cannot pass back an object of type \( \tau(CountingBag) \) in \( x \), i.e.,

\[
\text{Ptrans} \{ x : \tau(Bag) \}_{\text{st}} \{ x : \tau(Bag) \}_{\text{st}}
\]

is not a subtype of:

\[
\text{Ptrans} \{ x : \tau(CountingBag) \}_{\text{st}} \{ x : \tau(Bag) \}_{\text{st}}
\]

The object specification above consequently has limited applicability when it is not an object’s method. The object specification

\[
x.b::[x.b = \emptyset] \downarrow \text{Ptrans} \{ x : \tau(Bag) \}_{\text{st}} \{ x : \tau(Bag) \}_{\text{st}}
\]
should be compared with the (different) object specification:

\[ x.b :: [x.b = []] : \perp Ptrans \{ x : \tau(\text{CountingBag}) \}_{st} \{ x : \tau(\text{CountingBag}) \}_{st} \]

The latter object specification can return an object of type \( \tau(\text{CountingBag}) \) in the post-condition. The value of the \( cnt \) field will remain unchanged. To reiterate, the specification

\[ x.b :: [x.b = []] : \perp Ptrans \{ x : \tau(\text{Bag}) \}_{st} \{ x : \tau(\text{Bag}) \}_{st} \]

will accept a precondition predicate where \( x : \tau(\text{CountingBag}) \), but even though, dynamically \( x : \tau(\text{CountingBag}) \), statically:

\[ x.b :: [x.b = []] : \perp Ptrans \{ x : \tau(\text{Bag}) \}_{st} \{ x : \tau(\text{Bag}) \}_{st} \]

In other words, the definition of an object specification relies on a (static) (most defined) type constraint, rather than a dynamic type check. Consequently, the static type environment used to reason about object specifications is the most defined type.

Since object specifications are context sensitive to the (static) type environment, the rules that allow refinement of object specifications must also be context sensitive to the (static) type environment. The definition of refinement (Definition Refinement (4.32)) is context sensitive to the static type environment. Consequently, so are the refinement rules proved using that definition, e.g., Theorem Open World Specification (4.34).

The subtyping of methods in object types, discussed in Chapter 5, permits more flexibility, e.g.:

\[ Ptrans \{ x : \tau(\text{CountingBag}) \}_{st} \{ x : \tau(\text{CountingBag}) \}_{st} \cong \]
\[ Ptrans \{ x : \tau(\text{Bag}) \}_{st} \{ x : \tau(\text{Bag}) \}_{st} \]

When object-refining \( \text{Bag} \), the object specification \( x.b :: [x.b = []]_{x:\tau(\text{Bag})} \) could be refined to either

\[ x.b :: [x.b = []]_{x:\tau(\text{CountingBag})} \]

or

\[ x.b, \ x.cnt :: [x.b = [] \land x.cnt = 0]_{x:\tau(\text{CountingBag})} \]

using Theorem 7.16, or potentially, using Theorem 9.6, to:

\[ x.b :: [x.b = []]_{x.b | x.cnt} \ x.cnt :: [x.cnt = 0] \]

The definition of an object specification may appear to deny the modification of future, unknown attributes. The (static) most defined type defines the potential scope of the closed world. That is, any attribute in the most defined type that is not in the object specification’s frame, is placed under the closed world constraint, i.e., cannot be altered. For example, the object specification

\[ x.b :: [x.b = []]_{x:\tau(\text{Bag})} \]
in the environment \( x : \tau(Bag) \). In this environment, any attributes in the type \( \tau(Bag) \) other than \( b \) cannot be altered, i.e., they are under a closed world constraint. All (non) attributes outside of \( \tau(Bag) \) are unconstrained as they are outside the potential scope of the closed world. The (open world style) refinement relation allows the potential scope of the closed world to be increased by including new attributes. The actual scope of the closed world may or may not be enlarged, depending upon whether or not the new attributes are added to the frame of the object specification. If, during object-refinement, the new attributes are not included in the frame, then the definition of object specifications forces the new attributes under the closed world constraint, e.g., the attribute \( cnt \) cannot be altered in the object specification:

\[
x.b:: [x.b = []]
\]

In this instance, the potential scope of the closed world, as well as the actual scope of the closed world now both include the attribute \( cnt \). If, however, the new attributes are included in the frame then the potential scope of the closed world increases (with \( cnt \)) while the actual scope of the closed world does not, e.g., the following object specification allows the attribute \( cnt \) to be modified.

\[
x.b, x.cnt:: [x.b = []]_{\{x: \tau(CountingBag)\}_{ST}}
\]

Similarly, the potential scope of the closed world of the following object specification includes \( cnt \) while the actual scope does not. However, even though \( cnt \) is not in the closed world, it is still constrained by the postcondition.

\[
x.b, x.cnt:: [x.b = [] \land x.cnt = 0]_{\{x: \tau(CountingBag)\}_{ST}}
\]

### 7.3 Private Attributes

This section presents a model for private attributes. The use of private attributes effects a hiding of ‘state’. By hiding ‘state’, an object can be data refined without needing to modify its clients. Such data refinements are presented in Section 7.4.

Private attributes are attributes of an object that are inaccessible to clients of that object. They can be used for security (holding sensitive information) and for data-abstraction (withholding the data’s representation from the client). This allows the alteration of an object without affecting the object’s clients.
Private attributes can be modelled using subsumption. Given the object definition

\[
\begin{align*}
\text{var } o & := \text{object} \\
\text{field } f : F & := fv \\
\text{field } pf : PF & := pfv \\
\text{method } m & = mv \\
\text{method } pm & = pmv
\end{align*}
\]

\[\text{end \bullet} \]

\[\text{Prog} \]

a client can create instances of the object:

\[
\begin{align*}
\text{var } p & := \text{Object} \\
\text{field } f : F & \\
\text{field } pf : PF & \\
\text{method } m & \\
\text{method } pm & \\
\text{end} & := o \bullet
\end{align*}
\]

\[\text{SubProg} \]

\[\text{SubProg} \]

\[\text{SubProg} \]

\[\text{SubProg} \]

SubProg has access to all attributes of \( p \). If the syntax \( \tau(o) \) is used to denote the type of object \( o \) then the following program fragment is equivalent.

\[
\begin{align*}
\text{var } p & := \tau(o) := o \bullet \text{SubProg}
\end{align*}
\]

Alternatively, to model private attributes, certain attributes of \( p \) can be hidden by introducing \( p \) as a supertype of \( \tau(o) \):

\[
\begin{align*}
\text{var } p & := \text{Object} \\
\text{field } f : F & \\
\text{method } m & \\
\text{end} & := o \bullet
\end{align*}
\]

\[\text{SubProg} \]

Now SubProg only has access to attributes \( f \) and \( m \) of object \( p \). The advantage of hiding attributes of \( p \) is that the hidden attributes can, in future, be data refined without affecting SubProg.

To force all clients to introduce objects as supertypes of their actual types, the private syntax is used\(^3\).

\(^3\)For a different approach the reader is referred to the work of Sekerinski [Sek96] who uses existential types.
Definition 7.37 (Private Attribute)

\[
\begin{align*}
&\text{var } o := \text{object} \\
&\quad \text{field } f : F := fv \\
&\quad \text{private field } pf : PF := pfv \\
&\quad \text{method } m = mv \\
&\quad \text{private method } pm = pmv
\end{align*}
\]

\[
\text{end } \\
\text{Prog}\]

\[
\begin{align*}
&\text{var } o : \text{Object} \\
&\quad \text{field } f : F \\
&\quad \text{method } m
\end{align*}
\]

\[
\text{end} := \text{object} \\
\quad \text{field } f : F := fv \\
\quad \text{field } pf : PF := pfv \\
\quad \text{method } m = mv \\
\quad \text{method } pm = pmv
\]

\[
\text{end } \\
\text{Prog}\]

\[
\Box
\]

Clients of \( o \) can now only access the public attributes.

7.4 Data refinement for Objects

This section uses a specialisation of data refinement to provide an object-data-refinement relation that permits the data refinement of an object’s private data structures while requiring no change to the client’s code. After the object-data-refinement, the client can be modified to use the new attributes of the implementation object. The refinement of the client to use the new attributes of the implementation object is termed client enhancement.

Example 7.38 (Object-Data-Refinement Client Enhancement) The following is an example of the manner in which a client can be ‘enhanced’ subsequent to an object-data-refinement.

The example uses a specification object \( \text{Triangle} \), represented using (a sequence of) three points. Each point is represented as a pair of natural numbers, where the first element represents the x coordinate, and the second element of the pair represents the y coordinate. For example, \( \text{fst}(\text{points}(3)) \) represents the x coordinate of the third point, while \( \text{snd}(\text{points}(3)) \) represents the third point’s y coordinate. \( \text{Triangle} \) has an implicit invariant that restricts its first point to the origin, its second point to the y axis and its
CHAPTER 7. OBJECT- AND CLASS-REFINEMENT

third to the x axis, thereby forcing a right-angled triangle. The specification data type is illustrated in Figure 7.1. Triangle also has methods for setting the points and obtaining the distances between any two points.

Triangle :=
object
  private field points : seq_3(\mathbb{N} \times \mathbb{N}) := \{(0,0),(0,0),(0,0)\}
  method setPoints(value pnts : seq_3(\mathbb{N} \times \mathbb{N})) =
    \{fst(pnts(1)) = 0 = snd(pnts(1)) \land
    fst(pnts(2)) = 0 \land snd(pnts(3)) = 0\};
  points := pnts
  method getDistance(value p1 : 1..2, value p2 : 2..3, result distance : \mathbb{R}) =
    \{p1 \neq p2\};
  distance := \sqrt{(fst(points(p1)) - fst(points(p2)))^2 +
    (snd(points(p1)) - snd(points(p2)))^2}
end

Using an unspecified object-data-refinement, an implementation object ConcreteTriangle is calculated. As shown in Figure 7.2, the object ConcreteTriangle represents the resulting triangle using two lengths: the first being the height of the triangle and the second being the length of its base. It also has, for efficiency, a perimeter field and a new getPerimeter method.
object

private field height : \mathbb{N} := 0
private field width : \mathbb{N} := 0
private field perimeter : \mathbb{R} := 0

method setPoints(value pnts : seq_3(\mathbb{N} \times \mathbb{N})) =
    \{\text{fst}(pnts(1)) = 0 = \text{snd}(pnts(1)) \land
    \text{fst}(pnts(2)) = 0 \land \text{snd}(pnts(3)) = 0\};
    height := \text{snd}(pnts(2));
    width := \text{fst}(pnts(3));
    perimeter := \sqrt{\text{height}^2 + \text{width}^2 + \text{width} + \text{height}}

method getDistance(value p1 : 1..2, value p2 : 2..3, result distance : \mathbb{R}) =
    \{p1 \neq p2\};
    if (p1 = 1) then
        if (p2 = 2) then distance := height
        else distance := width;
    end
else
    distance := \sqrt{\text{height}^2 + \text{width}^2}
end

method getArea(result area : \mathbb{R}) =
    area := width * height / 2.0
method getPerimeter(result peri : \mathbb{R}) =
    peri := perimeter

end

Consequently, ‘old’ client code that calculates the perimeter could be refined to a call to
CHAPTER 7. OBJECT- AND CLASS-REFINEMENT

getPerimeter.

\[
\begin{align*}
&[\text{var} \ x, y, z : \mathbb{N} \bullet \\
&\quad o.\text{call} \ \text{getDistance}(1, 2, x) \\
&\quad o.\text{call} \ \text{getDistance}(2, 3, y) \\
&\quad o.\text{call} \ \text{getDistance}(1, 3, z) \\
&\quad p := x + y + z \\
&] \\
&\subseteq o.\text{call} \ \text{getPerimeter}(p)
\end{align*}
\]

When data refining a local variable block in the classical refinement calculus, only the code inside the local variable block has access to the specification or implementation variable. The specification/implementation variables are consequently inaccessible to, or hidden from, the code outside the variable block. In a similar fashion, one way to effect a data refinement of objects is to ensure that there is some state that is inaccessible to the client code. It is the modification of the content and behaviour of the inaccessible fields that forms the data refinement of objects. To effect a hiding of ‘state’, as seen in Section 7.3, the type of an object is masked by reporting to a client a supertype rather than the actual type.

Object-data-refinement is defined such that an object \( \text{spec} \) object-data-refines to an object \( \text{impl} \) if \( \text{impl} \) is formed by replacing some private fields of \( \text{spec} \), and data refining the methods of an object. A consistency check between the initial values of the implementation and specification variables is also required.

Object-data-refinement is a specialisation of local variable data refinement in which the implementation public type is a subtype of the specification public type (Line 7.1 below). This constraint allows specialised rules to be developed that allow the implementation client code to remain syntactically the same as the specification client code. After the object-data-refinement is completed, it is possible to further enhance the implementation client code using any new public attributes of the implementation object.

The definition of object-data-refinement requires the identification of the private attributes being removed (\( \text{PrivateSpec} \)) and those being added (\( \text{PrivateImp} \)). For the specification object \( \text{spec} \), the private attributes are those remaining after the public attributes (\( \tau_{\text{Public}}(\text{spec}) \)) have been ‘subtracted’ from the actual type. Alternatively, the private and public types should have no attributes in common, while together they should form the actual type:

\[
(\tau_{\text{Public}}(\text{spec}) \sqcup \text{PrivateSpec}) = \text{Top} \land (\tau_{\text{Public}}(\text{spec}) \sqcap \text{PrivateSpec}) = \tau(\text{spec})
\]

Similarly, using the implementation object \( \text{impl} \), the implementation private attributes \( \text{PrivateImp} \) can be identified:

\[
(\tau_{\text{Public}}(\text{impl}) \sqcup \text{PrivateImp}) = \text{Top} \land (\tau_{\text{Public}}(\text{impl}) \sqcap \text{PrivateImp}) = \tau(\text{impl})
\]
A predicate transformer rep must be chosen (Line 7.2 below) to provide a link between the public methods of the specification type and the corresponding methods of the implementation type. The definition requires all public fields of the specification type to object-refine to the corresponding implementation fields (Line 7.3). The definition also requires the private implementation fields to be initially set to a value allowed by the abstraction invariant (Line 7.4). To help define object-data-refinement, the following abbreviation is provided.

\[
\text{spec} \triangleleft_{\text{rep}} \text{impl} \triangleq \begin{align*}
\tau_{\text{Public}}(\text{impl}) &\preceq \tau_{\text{Public}}(\text{spec}) \land \\
\forall j \in \text{dom methods}(\tau_{\text{Public}}(\text{spec})) \bullet \text{spec}\dot{\circ} j &\preceq_{\text{rep}} \text{impl}\dot{\circ} j \land \\
\forall j \in \text{dom fields}(\tau_{\text{Public}}(\text{spec})) \bullet \text{spec}\dot{\circ} j &\subseteq^* \text{impl}\dot{\circ} j \land \\
\text{ai}
\end{align*}
\] (7.1)

(7.2)

(7.3)

(7.4)

**Definition 7.39 (Object-Data-Refinement)** Given value semantics objects spec and impl, an abstraction invariant ai that references the private specification and implementation attributes (through \(\text{self}_{\text{spec}}\) and \(\text{self}_{\text{impl}}\) respectively), \(\text{rep}'\) \(q \triangleq (\exists \text{self}_{\text{spec}} \bullet \text{spec} \triangleleft_{\text{rep}'} \text{impl} \land p)\), and

\[
\text{odrep} p \triangleq (\exists \text{self}_{\text{spec}} \bullet \text{spec} \triangleleft_{\text{rep}'} \text{impl} \land p)
\]

then object-data-refinement is defined as follows:

\[
\text{spec} \triangleleft_{\text{ai}} \text{impl} \triangleq \text{spec} \triangleleft_{\text{odrep}} \text{impl}
\]

This definition ensures that the public methods of spec are data refined to those of impl under the predicate transformer \(\text{odrep} p \equiv (\exists \text{self}_{\text{spec}} \bullet \text{spec} \triangleleft_{\text{rep}'} \text{impl} \land p)\). This constraint is necessary to ensure that all attributes of the object are mapped under the data-refinement relation. If object-data-refinement were defined instead as \(\text{spec} \triangleleft_{\text{rep}'} \text{impl}\), then a specification method could be data refined to an implementation method that arbitrarily modified the public attributes.

In a reference semantics program, the abstraction invariant \(\text{ai}\) references the private specification and implementation attributes through \(\text{store}_s(\text{self})\) and \(\text{store}_i(\text{self})\). The following definitions are analogous to those for \(\text{rep}'\) and \(\text{odrep}\).

\[
\text{rep}'' q \triangleq (\exists \text{store}_s \bullet \text{ai} \land \{\text{self}\} \triangleleft \text{store}_s = \{\text{self}\} \triangleleft \text{store}_i \land q)
\]

\[
\text{odrepr} p \triangleq (\exists \text{store}_s \bullet \text{spec} \triangleleft_{\text{rep}''} \text{impl} \land \{\text{self}\} \triangleleft \text{store}_s = \{\text{self}\} \triangleleft \text{store}_i \land p)
\]

The conjunct \(\{\text{self}\} \triangleleft \text{store}_s = \{\text{self}'\} \triangleleft \text{store}_i\) is included as only the object at reference self is to be altered. Object-data-refinement for reference semantics objects is defined as:

\[
\text{spec} \triangleleft_{\text{ai}} \text{impl} \triangleq \text{spec} \triangleleft_{\text{odrepr}} \text{impl}
\]

\(\text{This property is analogous to the constructor refinement criteria of Back, Mikhajlova and von Wright [BMvW00, p30].}\)
Example 7.40 (An Object-Data-Refinement) Assume \( \text{inc}_{\text{spec}} \) is an object with a private field \( m : \mathbb{Z} \), and public methods \( \text{inc} \) and \( \text{result} \) that are used to increment \( m \) by two, and return half of \( m \), respectively.

\[
\text{inc}_{\text{spec}} \triangleq \begin{align*}
\text{object} \\
\text{private field } m : \mathbb{Z} := 2 \times \alpha, \\
\text{method } \text{inc} = \text{self} \uparrow m := \text{self} \uparrow m + 2, \\
\text{method } \text{result}(\text{result } r : \mathbb{Z}) = r := \text{self} \uparrow m / 2
\end{align*}
\]

End

Assume \( \text{inc}_{\text{impl}} \) has a private field \( n : \mathbb{Z} \) and public methods \( \text{inc} \) and \( \text{result} \) that are used to increment \( n \) by one, and return \( n \), respectively.

\[
\text{inc}_{\text{impl}} \triangleq \begin{align*}
\text{object} \\
\text{private field } n : \mathbb{Z} := \alpha, \\
\text{method } \text{inc} = \text{self} \uparrow n := \text{self} \uparrow n + 1, \\
\text{method } \text{result}(\text{result } r : \mathbb{Z}) = r := \text{self} \uparrow n
\end{align*}
\]

The reference semantics object \( \text{inc}_{\text{spec}} \) object-data-refines to \( \text{inc}_{\text{impl}} \) under \( ai \triangleq \text{inc}_{\text{spec}} \uparrow o m = 2 \times \text{inc}_{\text{impl}} \uparrow o n \) provided the value of \( \text{inc}_{\text{impl}} n \) is half that of \( \text{inc}_{\text{spec}} m \). Line 7.1 is valid as both objects only have public methods \( \text{inc} \) and \( \text{result} \). Line 7.2 is valid as the methods data refine:

\[
\begin{align*}
\text{self} \uparrow n & := \text{self} \uparrow n + 1 \\
\text{rep} \triangleq r & := \text{self} \uparrow n
\end{align*}
\]

Line 7.3 is vacuously true. Line 7.4 reduces to the abstraction invariant \( \text{spec} \uparrow o m = 2 \times \text{impl} \uparrow o n \) which is the proviso identified earlier.

\[\Diamond\]

### 7.4.1 Simulation of Object-Data-Refinements

This section presents a rule and corresponding proof that allows the simulation of an object by an object-data-refinement.
Theorem 7.41 (Object Simulation/Polyomorphism) A specification object \( \textit{spec} \), an implementation object \( \textit{impl} \), and an abstraction invariant \( \textit{ai} \) are assumed such that \( \textit{spec} \preceq_{\textit{ai}} \textit{impl} \). Additionally, it is assumed that \( \textit{store}_s \) holds instances of \( \tau_{\text{Public}}(\textit{spec}) \), \( \textit{store}_i \) holds instances of \( \tau_{\text{Public}}(\textit{impl}) \), and objects \( \textit{e}_{1..n} \) are object assigned to \( \textit{o} \) within \( \textit{Prog}(\textit{e}_1, \ldots, \textit{e}_j, \ldots, \textit{e}_n) \). Given objects \( \textit{f}_{1..n} \) such that \( \forall j \in 1..n \bullet \textit{e}_j \preceq_{\textit{ai}} \textit{f}_j \):

\[
\begin{align*}
\text{\textit{o} := new spec; Prog(\textit{e}_1, \ldots, \textit{e}_j, \ldots, \textit{e}_n)}
\end{align*}
\]

\[
\begin{align*}
\text{\textit{o} := new impl; Prog(\textit{f}_1, \ldots, \textit{f}_j, \ldots, \textit{f}_n)[\textit{store}_s\backslash\textit{store}_i]}
\end{align*}
\]

Proof

The proof is achieved by data refining the store so that the object at location \( \textit{o} \) is object-data-refined: \( \textit{o}^\uparrow_{\textit{store}_s} \preceq_{\textit{ai}} \textit{o}^\uparrow_{\textit{store}_i} \). The store is therefore data refined under \( \textit{REP} \):

\[
\begin{align*}
\textit{REP} \; p \; \triangleq \; (\exists \textit{store}_s \bullet \textit{o}^\uparrow_{\textit{store}_s} \preceq_{\textit{ai}} \textit{o}^\uparrow_{\textit{store}_i} \land (\{\textit{o}\} \ll \textit{store}_s) = (\{\textit{o}\} \ll \textit{store}_i) \land p )
\end{align*}
\]

The \textit{new} construct data refines to the implementation version using Data Refine New (7.42) which is the next theorem presented.

\[
\textit{o} := \textit{new spec} \preceq_{\textit{REP}} \textit{o} := \textit{new impl}
\]

Consequently, the proof reduces to showing that the remainder of the code, \( \textit{Prog}(\textit{e}_1, \ldots, \textit{e}_j, \ldots, \textit{e}_n) \), piecewise data refines to itself (under the appropriate substitution \( [\textit{store}_s\backslash\textit{store}_i] \)):

\[
\begin{align*}
\textit{Prog}(\textit{e}_1, \ldots, \textit{e}_j, \ldots, \textit{e}_n) \preceq_{\textit{REP}} \textit{Prog}(\textit{f}_1, \ldots, \textit{f}_j, \ldots, \textit{f}_n)[\textit{store}_s\backslash\textit{store}_i]
\end{align*}
\]

It is easily shown that the language constructs that do not involve dereferences of \( \textit{o} \) data refine to themselves given the partial abstraction invariant:

\[
\begin{align*}
(\{\textit{o}\} \ll \textit{store}_s) = (\{\textit{o}\} \ll \textit{store}_i)
\end{align*}
\]

This leaves the statements that involve \( \textit{o}^\uparrow \). The data refinement rules following Theorem 7.42 demonstrate that statements involving \( \textit{o}^\uparrow \) data refine to themselves.

\[\text{QED}\]

Theorem 7.42 (Data Refine New) Proof on page 227 Given \( \textit{spec} \preceq_{\textit{ai}} \textit{impl} \) and

\[
\begin{align*}
\textit{REP} \; p \; \triangleq \; (\exists \textit{store}_s \bullet \textit{o}^\uparrow_{\textit{store}_s} \preceq_{\textit{ai}} \textit{o}^\uparrow_{\textit{store}_i} \land (\{\textit{o}\} \ll \textit{store}_s) = (\{\textit{o}\} \ll \textit{store}_i) \land p )
\end{align*}
\]

then

\[
\begin{align*}
\textit{o} := \textit{new spec} \preceq_{\textit{REP}} \textit{o} := \textit{new impl}
\end{align*}
\]
CHAPTER 7. OBJECT- AND CLASS-REFINEMENT

Theorem 7.43 (Data Refine Store in Method Call)  Proof on page 230  Given public method \( m \), and
\[
\text{REP } p \models (\exists \text{store}_s \bullet o \uparrow_{\text{store}} \preceq_{\text{ai}} o \uparrow_{\text{store}} \land (\{o\} \preceq \text{store}_s) = (\{o\} \preceq \text{store}_i) \land p)
\]
then
\[
o \uparrow_{\text{store}} \cdot \text{call } m \preceq_{\text{REP}} o \uparrow_{\text{store}} \cdot \text{call } m
\]

Theorem 7.44 (Data Refine Object Assignment)  Proof on page 231  Given \( e \preceq_{\text{ai}} f \), and
\[
\text{REP } p \models (\exists \text{store}_s \bullet o \uparrow_{\text{store}} \preceq_{\text{ai}} o \uparrow_{\text{store}} \land (\{o\} \preceq \text{store}_s) = (\{o\} \preceq \text{store}_i) \land p)
\]
then
\[
o \uparrow_{\text{store}} := e \preceq_{\text{REP}} o \uparrow_{\text{store}} := f
\]

Example 7.45 (An Object Simulation)  Assume objects \( \text{inc}_{\text{spec}} \) and \( \text{inc}_{\text{impl}} \) from Example An Object-Data-Refinement (7.40) where \( \text{inc}_{\text{spec}} \preceq_{\text{ai}} \text{inc}_{\text{spec}} \otimes_{n=\text{inc}_{\text{impl}}} \text{inc}_{\text{impl}} \). The following refinement holds in a context with variable \( p \) of type \( \mathbb{Z} \).

\[
\begin{align*}
&\left[ [\text{var } o : \text{Ref } \{\text{store}_s\} \bullet o := \text{new } \text{inc}_{\text{spec}}; o \uparrow_{\text{store}} \cdot \text{call } \text{inc}; o \uparrow_{\text{store}} \cdot \text{call } \text{result}(p) ] \right] \\
&\subseteq \text{Object Simulation/Polymorphism (7.41)} \\
&\left[ [\text{var } o : \text{Ref } \{\text{store}_i\} \bullet o := \text{new } \text{inc}_{\text{impl}}; o \uparrow_{\text{store}} \cdot \text{call } \text{inc}; o \uparrow_{\text{store}} \cdot \text{call } \text{result}(p) ] \right]
\end{align*}
\]
\(
\checkmark
\)

Although data refinement of objects can be effected using the rules presented in Chapter 2, additional support is provided for object specifications in the following theorem.

Theorem 7.46 (Data Refine Object Specifications (Semantics for Values))  Proof on page 233  Assume an abstract object \( \text{spec} \), a concrete object \( \text{impl} \), abstraction invariant \( AI \), abstract fields in the frame \( af \), abstract fields not in the frame \( anf \), introduced concrete fields in the frame \( bf \), concrete fields not in the frame \( bnf \), common fields in the frame \( gf \), and common fields not in the frame \( gnf \). Given \( ai \equiv AI \land spec.gf \subseteq^\ast impl.gf \land spec.gnf \subseteq^\ast impl.gnf \), and \( \text{REP } p \equiv (\exists \text{spec } \bullet ai \land p) \), an object specification data refines as follows:

\[
spec.af, spec.gf :: [\text{pre }, \text{post}] \\
\preceq_{\text{REP}} \\
\left[ [\text{con } spec_0 ] \bullet \\
\text{impl.bf } ; :: [ \text{ (AI)[spec\backslash spec_0] } \land spec_0.gf \subseteq^\ast \text{impl.gf } \land \text{spec_0.gnf } \subseteq^\ast \text{impl.gnf } \land (\text{pre}[spec\backslash spec_0] ) ] \\
\text{impl.gf } ; :: [ \text{spec.gnf } \subseteq^\ast \text{impl.gnf } \land \text{post } \land \text{spec_0.gnf } \subseteq^\ast \text{spec.gnf } ]
\]
\]
provided

\[
AI_0 \Rightarrow (AI \land (\text{impl}_0 \sqsubseteq \text{impl}.bnf) \Rightarrow (\text{spec}_0 \sqsubseteq \text{spec}.anf))
\]

where

\[
AI_0 \equiv (AI)[\text{spec}, \text{impl}_0 \setminus \text{spec}_0, \text{impl}_0]
\]

**Example 7.47 (Data Refine Bag)** Assume the following object.

\[
\begin{align*}
\text{Bag} & \triangleq \text{object} \\
\text{private field } b &: \text{bag Char} := [] \\
\text{method } \text{Add}(\text{value } c &: \text{Char}) & \triangleq \\
& \text{self}.b :: [\text{self}.b = \text{self}_0.b \uplus [c]] \\
\text{method } \text{Choose}(\text{result } c &: \text{Char}) & \triangleq \\
& \text{self}.b, c :: [\text{self}.b \neq [], \text{self}_0.b = \text{self}.b \uplus [c]] \\
\text{method } \text{AddAll}(\text{value } nb &: \text{Bag}) & \triangleq \\
& \text{self}.b :: [\text{self}.b = \text{self}_0.b \uplus nb.b] \\
\text{method } \text{Empty}(\text{result } e &: \mathbb{B}) & \triangleq \\
& e :: [e = (\text{self}.b = [])]
\end{align*}
\]

end

The bag type field \( b \) can be data refined to a function (array). Assume the concrete field \( \text{self'} . bi : \text{Char} \rightarrow \mathbb{N} \) and the abstraction invariant

\[
ai \equiv \text{self'} . bi = \{ j \in \text{self}.b \mid j \mapsto \text{self}.b[j] \}
\]

where ‘in’ is the ‘in bag’ operator and \( b[j] \) returns the number of elements \( j \) in bag \( b \). Consequently, \( \text{Bag} \) can be object-data-refined under \( ai \) to \( \text{BagImp} \), \( \text{Bag} \triangleright_{ai} \text{BagImp} \).
where

\[
\begin{align*}
\text{BagImp} & \equiv \text{object} \\
\text{private field } bi : \text{Char} & \rightarrow \mathbb{N} := \{\} \\
\text{method } \text{Add(value } c : \text{Char) } & \equiv \\
& \left[ \begin{array}{l}
\text{var bagdom : P Char} \\
\text{bagdom := dom(self.bi);} \\
\text{if } c \in \text{bagdom} \rightarrow \\
\quad \text{self.bi}(c) := \text{self.bi}(c) + 1 \\
\text{else} \\
\quad \text{self.bi}(c) := 1
\end{array} \right]
\end{align*}
\]

\[
\begin{align*}
\text{method } \text{Choose(result } c : \text{Char) } & \equiv \\
& \left[ \begin{array}{l}
\text{var cnt : N} \\
\text{cnt := self.bi(c);} \\
\text{if } \text{cnt} > 1 \rightarrow \\
\quad \text{self.bi}(c) := \text{self.bi}(c) - 1 \\
\text{else} \\
\quad \text{self.bi} := \{c\} \leftarrow \text{self.bi}
\end{array} \right]
\end{align*}
\]

\[
\begin{align*}
\text{method } \text{AddAll(value } nb : \text{Bag) } & \equiv \\
& \left[ \begin{array}{l}
\text{var nbcopy : } \tau(\text{Bag}), e : \text{Bag} \\
\text{nbcopy := nb;} \\
\text{nbcopy.call Empty}(e); \\
\text{do } e \rightarrow \\
\quad \left[ \begin{array}{l}
\text{var } c : \text{Char} \\
\text{nbcopy.call Choose}(c); \\
\text{self.call Add}(c)
\end{array} \right] \\
\text{nbcopy.call Empty}(e)
\end{array} \right]
\end{align*}
\]

\[
\begin{align*}
\text{method } \text{Empty(result } e : \text{Bag) } & \equiv \\
& e := (\text{self.bi} = \{\})
\end{align*}
\]

The introduction of the \text{Empty} method was used to avoid the need to access \text{nbcopy}'s private variable \( b \) as seen in the body of \text{AddAll} in Example 7.36.

To establish \( \text{Bag} \preceq_{\text{ai}} \text{BagImp} \) the type of the implementation must be a subtype of that of the specification (i.e., \( \tau(\text{BagImp}) \preceq \tau(\text{Bag}) \)), and the initialisation of the abstract field must map under the abstraction invariant to the initialisation of the concrete field.
That is, assuming \( \text{self'}.bi = [ ] \), then:

\[
\begin{align*}
ai & \equiv \text{self'}.bi = \{ j \in \text{self}.b \cdot j \mapsto \text{self}.b[j] \} \\
\equiv \{ \} & = \{ j \in [ ] \cdot j \mapsto [ [ ]][j] \}
\end{align*}
\]

which holds.

The object-data-refinement also requires that the methods be data refined under \( \text{odrep} \)
where \( \text{rep'} \ q \sqsubseteq (\exists \text{self} \cdot ai \wedge q) \), and

\[
\text{odrep} \ p \sqsubseteq (\exists \text{self} \cdot \text{spec} \ll_{\text{rep'} impl} \wedge p)
\]

The method \( \text{Add} \) is shown to data refine under \( \text{rep'} \) as follows.

\[
\begin{align*}
\text{self}.b:: & \left[ \text{self}.b = \text{self}_0.b \uplus [c] \right] \\
\prec_{\text{rep'}} & \text{Data Refine Object Specifications (Semantics for Values) (7.46)} \\
\left[ \begin{array}{c}
\text{con} \text{self}_0 \cdot \\
\end{array} ight] & \\
\text{self'}.bi:: & \left[ \begin{array}{c}
\text{self'}.bi = \{ j \in \text{self}_0.b \cdot j \mapsto \text{self}_0.b[j] \} \\
(\exists \text{self} \cdot \text{self'}.bi = \{ j \in \text{self}.b \cdot j \mapsto \text{self}.b[j] \}) \land \\
\text{self}.b = \text{self}_0.b \uplus [c] \right] \\
\left[ \begin{array}{c}
\text{con} \text{self}_0 \cdot \\
\end{array} ight] & \\
\text{self'}.bi:: & \left[ \begin{array}{c}
\text{self'}.bi = \{ j \in (\text{self}_0.b \uplus [c]) \cdot j \mapsto (\text{self}_0.b \uplus [c])[j] \} \\
\end{array} \right]
\end{align*}
\]

The proviso of this step reduces to \( \text{true} \) as the only abstract field is in the frame.

The existential quantification is removed once all occurrences of \( \text{self} \) are replaced.

\[
\Box \text{Object Specification Strengthen Postcondition (7.19)} \\
\left[ \begin{array}{c}
\text{con} \text{self}_0 \cdot \\
\end{array} \right] & \\
\text{self'}.bi:: & \left[ \begin{array}{c}
\text{self'}.bi = \{ j \in \text{self}_0.b \cdot j \mapsto \text{self}_0.b[j] \} \\
(\exists \text{self} \cdot \text{self'}.bi = \{ j \in (\text{self}_0.b \uplus [c]) \cdot j \mapsto (\text{self}_0.b \uplus [c])[j] \}) \\
\end{array} \right]
\]

An alternation is introduced based on whether or not the function already has elements \( c \).

A local variable \( \text{bagdom} \) is introduced as alternation branches cannot include objects as

discussed on page 84.

\[
\Box \text{Introduce Local Variable Block (B.55)} \\
\text{Introduce Sequential Composition (Semantics for Values) (7.21)} \\
\text{Introduce Assignment (B.32)} \\
\text{Object Specification Alternation (7.22)} \\
\left[ \begin{array}{c}
\text{con} \text{self}_0 \cdot \\
\end{array} \right] & \\
\left[ \begin{array}{c}
\text{var} \text{bagdom} : \mathbb{P} \text{Char} \cdot \\
\text{bagdom} := \text{dom(}\text{self'}.bi); \\
\text{if} \ c \in \text{bagdom} \rightarrow \\
\text{self'}.bi:: & \left[ \begin{array}{c}
\text{self'}.bi = \{ j \in \text{self}_0.b \cdot j \mapsto \text{self}_0.b[j] \} \\
(\exists \text{self} \cdot \text{self'}.bi = \{ j \in (\text{self}_0.b \uplus [c]) \cdot j \mapsto (\text{self}_0.b \uplus [c])[j] \}) \\
\end{array} \right] \\
\text{else} & \\
\text{self'}.bi:: & \left[ \begin{array}{c}
\text{self'}.bi = \{ j \in (\text{self}_0.b \uplus [c]) \cdot j \mapsto (\text{self}_0.b \uplus [c])[j] \} \\
\end{array} \right]
\right. \\
\left. \begin{array}{c}
\text{fi} \\
\end{array} \right]
\]
\]
The postconditions are now strengthened using the guard information.

\[
\begin{align*}
&\equiv \text{Object Specification Strengthen Postcondition (7.19)} \\
&\text{Object Specification Weaken Precondition (7.18)} \\
&\text{Remove Logical Constant (B.45)} \\
&\left[ \begin{array}{l}
\text{var } \text{bagdom : P Char} \\
\text{bagdom} := \text{dom}(\text{self'.bi}); \\
\text{if } c \in \text{bagdom} \rightarrow \\
\quad \text{self'.bi}:: [\text{self'.bi} = \text{self}_0'.\text{bi} \oplus \{c \mapsto (\text{self'.bi}(c) + 1)\}] \\
\text{else} \\
\quad \text{self'.bi}:: [\text{self'.bi} = \text{self}_0'.\text{bi} \cup \{c \mapsto 1\}] \\
\text{fi} \\
\end{array} \right]
\]

Both specifications can be refined to field updates.

\[
\begin{align*}
&\equiv \text{Introduce Field Update (Semantics for Values) (7.29)} \\
&\left[ \begin{array}{l}
\text{var } \text{bagdom : P Char} \\
\text{bagdom} := \text{dom}(\text{self'.bi}); \\
\text{if } c \in \text{bagdom} \rightarrow \\
\quad \text{self'.bi} := \text{self'.bi} \oplus \{c \mapsto (\text{self'.bi}(c) + 1)\} \\
\text{else} \\
\quad \text{self'.bi} := \text{self'.bi} \cup \{c \mapsto 1\} \\
\text{fi} \\
\end{array} \right]
\]

Using the accessed function syntax results in the following code.

\[
\begin{align*}
&\equiv \text{Accessed Function Assignment (6.12)} \\
&\left[ \begin{array}{l}
\text{var } \text{bagdom : P Char} \\
\text{bagdom} := \text{dom}(\text{self'.bi}); \\
\text{if } c \in \text{bagdom} \rightarrow \\
\quad \text{self'.bi}(c) := \text{self'.bi}(c) + 1 \\
\text{else} \\
\quad \text{self'.bi}(c) := 1 \\
\text{fi} \\
\end{array} \right]
\]

The remaining methods are assumed to similarly data refine. To complete the proof, the methods are shown to data refine under odrep. The steps are almost exactly the same, except that the equivalents of Lines 7.1, 7.2, and 7.3 of Definition Object-Data-Refinement (7.39) are added to the precondition and postcondition of the specification on Line 7.5. These conjuncts are trivially removed using Strengthen Postcondition (B.54) and Weaken Precondition (B.53) as there are no public methods, and the methods data refine under rep'.

\diamond
CHAPTER 7. OBJECT- AND CLASS-REFINEMENT

7.5 Class-Based Refinement

This section details the extension of the object-based refinement calculus to a class-based refinement calculus. Several principles are used to guide the class-based semantics.

- Classes are defined as objects whose main purpose is the creation of class instances by cloning.

- Subclassing is a mechanism for incremental extension of classes. Subclassing is regarded as the class equivalent of subtyping, i.e., syntactic conformance to a signature (but not necessarily behavioural consistency—the field and method types conform but the methods need not be refinements).

- Finally, class refinement is subclassing with an additional behavioural consistency constraint. This split approach to behavioural consistency permits additional freedom during specification, yet allows the refiner to maintain the integrity of critical class sub-hierarchies.

The class-based refinement calculus defined here is a definitional extension of the object-based refinement calculus.

Definition 7.48 (Classes) A class is defined as an object.

\[\llbracket\text{class } \text{classname is}\]
\[
\begin{align*}
\text{field } f_{i \in 1..n} &: F_i := f v_i \\
\text{method } m_{i \in 1..m} &= m v_i \\
\end{align*}
\]
\end{align*}
\]
\]

\[\text{classname} := \text{object}\]
\[
\begin{align*}
\text{field } f_{i \in 1..n} &: F_i := f v_i \\
\text{method } m_{i \in 1..m} &= m v_i \\
\end{align*}
\]
\]

Classes defined in this fashion can be instantiated using the same mechanism that allows cloning in the object-based refinement calculus. That is, class instantiation for a semantics for values is achieved by object assignment or, for a semantics for references, reference cloning.
A class may be refined either through object-refinement or object-data-refinement. The introduction of a subclass that is also a class refinement maintains the behavioural consistency of the original class hierarchy.

The refinement rules of the object-based refinement calculus are upheld for classes. Additional rules, such as Class Introduction (7.49) can also be used to support class-based development methods.

**Theorem 7.49 (Class Introduction)**  
Proof on page 235  
This refinement rule can be used to introduce a new class into an existing class hierarchy. Since a class definition is merely a scoped variable introduction with a specific object initialisation, similar rules apply to class introduction as to variable introduction. Namely, a class with class name \( \text{Classname} \) can be introduced into a program \( P \) provided \( \text{Classname} \) is not free in \( P \):

\[
\text{Classname} \ nfi \ P
\]

\[
P \sqsubseteq \left[ \text{class} \ \text{Classname} \ \text{is} \ \text{Attribs} \ \text{end} \bullet P \right]
\]

where \( \text{Attribs} \) is a list of attributes.

The previously presented object-refinement rules can be lifted to classes. For instance, Object-Refine in Assignment (Semantics for Values) (7.11) can be lifted as shown in the following theorem.

**Theorem 7.50 (Refine Classes)**  
Classes can be refined if their associated objects object-refine. Given program \( P \),

\[
\text{object} \ \text{Attribs} \ \text{end} \sqsubseteq^* \text{object} \ \text{Attribs'} \ \text{end}
\]

\[
\left[ \text{class} \ \text{Classname} \ \text{is} \ \text{Attribs} \ \text{end} \bullet P \right] \sqsubseteq \left[ \text{class} \ \text{Classname} \ \text{is} \ \text{Attribs'} \ \text{end} \bullet P \right]
\]

**Proof**  
The proof is a straightforward application of Object-Refine in Assignment (Semantics for Values) (7.11) and Classes (7.48).

**QED**

Like classes, subclasses are also defined as objects.

**Definition 7.51 (Subclasses)**  
Subclassing updates the original class with new or overrid-
den attributes.

\[
\begin{align*}
&\text{subclass subclassname of classname is} \\
&\quad \text{field } f_{i \in 1..fn} : F_i := f_{vi} \\
&\quad \text{method } m_{i \in 1..mn} = m_{vi} \\
&\quad \text{end } \bullet \\
&\quad \text{Prog}
\end{align*}
\]

\[
\begin{align*}
\vDash
\begin{array}{l}
\quad \text{Var subclassname : } \tau((\text{classname} \circ f_{i \in 1..fn} \leftarrow f_{vi}) \circ m_{i \in 1..mn} \leftarrow m_{vi}) \bullet \\
\quad \text{subclassname} := (\text{classname} \circ f_{i \in 1..fn} \leftarrow f_{vi}) \circ m_{i \in 1..mn} \leftarrow m_{vi} ; \\
\quad \text{Prog}
\end{array}
\end{align*}
\]

for fresh subclassname.

\begin{itemize}
\item Like classes, subclasses may also be refined.
\end{itemize}

**Theorem 7.52 (Refine Subclasses)** Subclasses can be refined if their associated objects object-refine. Given program \( P \),

\[
\begin{align*}
\text{object } \text{Attribs end} \sqsubseteq^c \\
\text{object } \text{Attribs'} \text{ end}
\end{align*}
\]

\[
\begin{align*}
\vDash
\begin{array}{l}
\quad \text{subclass subclassname of classname is} \quad \text{Attribs end } \bullet \quad \text{Prog} \\
\end{array}
\end{align*}
\]

**Proof**
The proof is a straightforward application of Object-Refine in Assignment (Semantics for Values) (7.11) and Subclasses (7.51).

\( QED \)

### 7.6 Summary

This chapter has introduced an object-refinement relation, an object-data-refinement relation, and associated refinement rules. The main results of the chapter are Theorem 7.27 and Corollary 7.28 which allow an object instantiation to be refined to an instantiation of an object-refinement. Finally, the chapter lifted the calculus to a class-based refinement calculus.
Chapter 8

Towards A Reference Semantics

Object identity is an important aspect of object-oriented programming. Object-identities are used to ease the dynamic construction of object hierarchies that mirror complex, real-world entity interactions.

A natural way to represent object identities is to associate each object with a reference. For example, in a specification of a library, each book on the library’s shelf would be assigned a unique reference. Consequently, copies of the same book would be associated with different references. To access or modify the object, the reference can be used. Section 6.2 introduced a store that provided a mapping from references to their associated objects. This chapter introduces several novel techniques that ease the burden of reasoning about refinement calculus programs that use a store. A problem that occurs with the use of a store is that two variables may contain the same reference. This is termed aliasing. The problem with aliasing is that the modification via one variable will alter the value accessible via an alias. Some of the problems with aliasing are as follows.

- A seemingly innocuous predicate may become unsatisfiable in the presence of aliasing. For example, establishing $\alpha = \beta + 1$ is trivial when $\alpha$ and $\beta$ are not aliased, but when they are aliased, it is equivalent to establishing $\alpha = \alpha + 1$ or $False$. Only magic can establish $False$.

- A property involving one variable can be violated by the alteration of an aliased variable. For example, if $\alpha$ and $\beta$ are aliased and $\beta = 0$, then modifying $\alpha$ to achieve $\alpha = 1$ will falsify the original predicate $\beta = 0$.

- A property may already hold even though it may not appear to. For example, given $\beta = 0$, then one may try to modify $\alpha$ so that $\alpha = 0$ when it already does—as $\alpha$ and $\beta$ are aliased.

Aliasing manifests itself in a variety of means. For example, aliases may be introduced by assigning one reference to another. Alternatively, passing the same variable twice to a procedure as a reference parameter means that there are two different variables, that are aliased. Additionally, passing a global variable to a procedure as a reference parameter
also constructs an alias. Morgan [Mor88] discusses the various means by which such aliasing can be avoided.

Utting [Utt92, Utt97] has investigated the incorporation into the refinement calculus of the store approach to object identities. He showed that a linear increase in the number of reference values (variables) results in an exponential increase in the number of aliasing checks required to reason about references. For example, in an environment with three references, \(\alpha, \beta, \gamma : \text{Ref}\), to discover the value of an expression \(E\) after variable modification, five different alias partitions must be considered: \(\{\alpha, \beta, \gamma\}, \{\alpha, \beta\gamma\}, \{\alpha, \beta\gamma\}, \{\alpha \gamma, \beta\}, \{\alpha \beta \gamma\}\). In an environment with four variables, \(\alpha, \beta, \gamma, \delta\), fifteen different alias partitions must be considered: \(\{\alpha, \beta, \gamma, \delta\}, \{\alpha, \beta\gamma, \delta\}, \{\alpha, \beta\gamma, \gamma\}, \{\alpha, \beta, \gamma\delta\}, \{\alpha, \beta, \gamma, \delta\}, \{\alpha, \beta, \gamma, \beta\delta\}, \{\alpha, \beta, \gamma, \beta\gamma\}, \{\alpha, \beta, \gamma, \delta\}, \{\alpha, \beta, \gamma, \beta\delta\}, \{\alpha, \beta, \gamma, \beta\gamma\}, \{\alpha, \beta, \gamma, \delta\}, \{\alpha, \beta, \gamma, \beta\delta\}, \{\alpha, \beta, \gamma, \beta\gamma\}, \{\alpha, \beta, \gamma, \delta\}, \{\alpha, \beta, \gamma, \beta\delta\}, \{\alpha, \beta, \gamma, \beta\gamma\}, \{\alpha, \beta, \gamma, \delta\}\).

Many (non-object-oriented) languages use the types of references to curtail the exponential explosion. For example, if the types of two references are different then the references cannot be aliased. Unfortunately this does not hold for languages which allow subtyping, or the assignment of heterogeneous types. Despite this, the subtyping hierarchy for languages that do not support multiple inheritance can be used to remove the need for many alias checks. That is, if a reference is of a type that is not a sub- or super-type of the type of another reference, then provided subsumption is the only form of heterogeneous type assignment in the language these two references cannot be aliases.

Utting also split the store into smaller, local stores. Since references in different local stores cannot alias each other, alias checks between such references are not required. He introduced local stores that could contain heterogeneous types and a transfer operation which allowed references in one local store to be transferred to another. Finally, he showed that a program implemented using a number of local stores can be data refined to a program which uses a single, global store. This data refinement transforms the transfer operation to \texttt{skip}.

Other (refinement calculus) related work includes that of Bancroft [Ban97], who presents an abstract syntax, static semantics and dynamic environment, for a language containing both references and type extension. Butler [But99] has produced a technique for reasoning about trees that are implemented using pointers. Despite this work, the construction of a program that uses references still requires tedious ‘array-like’ reasoning. Even examples such as that by Bancroft (a linked list implementation of a sequence) that do not involve any aliasing (there is only one path to any element in the linked list) are quite complicated\footnote{Utting’s [Utt97, p9] linked list implementation of a sequence has an alias to the tail of the linked list, paraphrased as such: \texttt{head.next\textsuperscript{num} = tail} where \texttt{num} is the number of elements in the linked list and \texttt{next\textsuperscript{num}} is the dereferencing of \texttt{head} through field \texttt{next num} times.}

When developing programs that use references, the reasoning is verbose. To improve the clarity and conciseness of the reasoning, several syntactic shorthands are introduced
in this chapter. Targeting the syntactic representation of aliasing information, Section 8.1 introduces a variable aliasing annotation. This syntax is also designed to aid the collection and distribution of information about aliases and non-aliases. This approach differs to that of Utting who focuses on the development of specifications in a context where no aliasing information is known.

Besides aliasing constraints, specifications involving references are also cluttered with predicates involving constraints on the store. To ease the burden of writing and understanding specifications involving references, Section 8.2 presents the syntax and semantics of reference specifications. Reference specifications incorporate annotations on the frame variables of a classical specification statement to denote certain constraints on the store.

A general proof technique is discussed in Section 8.3. This technique involves data refining a program to an analogous program in a simpler environment. The two programs are related by a predicate transformer that is universal join homomorphic\(^2\). That is, the original program in a complex environment is ‘projected’ into the less complex environment. After development in this temporary environment, an inverse of the first data refinement is used to transform the program back to its original environment. This technique is applied in Section 8.4 to remove superfluous aliasing. This involves an innovative technique in which aliased variables are coalesced, allowing all but one of the aliased variables to be removed. Data refinement via inverse commands is also used in Section 8.5 in which an original technique is introduced to transform a reference semantics program into a corresponding value semantics program and vice versa. A corollary of this technique is that programs or program segments involving references that do not require aliasing can be developed within a value semantics and automatically converted into an analogous reference semantics program or program segment. A linked list implementation, similar to that of Bancroft and Utting, is developed in Section 8.6 using the techniques discussed.

8.1 Aliasing Annotations

A syntactic aliasing annotation that provides a shorthand notation for alias and non-alias information is introduced here. One way of representing the aliasing information of a program is to construct a set of possible partitions (\(\theta\)) of variable names. For example, given references \(\alpha\) through \(\gamma\) and no known aliasing information, the set of possible alias combinations is:

\[
\theta \equiv \{ \{\alpha, \beta, \gamma\}, \{\alpha, \beta \gamma\}, \{\alpha \beta, \gamma\}, \{\alpha \gamma, \beta\}, \{\alpha \beta \gamma\} \}
\]

If it is known that \(\alpha\) and \(\beta\) are aliases, this set can be reduced to:

\[
\theta \equiv \{ \{\alpha \beta, \gamma\}, \{\alpha \beta \gamma\} \}
\]

\(^2\)As defined in Desideratum 2.6.
This representation of aliasing information is monolithic yet easy to understand. However, in practice this representation is difficult to use. A distributed, variable focused representation is introduced here. The new representation can be viewed as constraining the possible sets in $\theta$.

**Definition 8.1 (Aliased)** Given variable sets $\bar{x}$ and $\bar{y}$, the syntax $(\alpha \approx_{\bar{x}}^{\bar{y}})$ denotes that the variable $\alpha$ is aliased with all the variables in the set $\bar{x}$ and is not aliased to any variable in $\bar{y}$.

\[
(\alpha \approx_{\bar{x}}^{\bar{y}}) \equiv (\forall i \in \bar{x} \bullet \alpha = i) \land (\forall j \in \bar{y} \bullet \alpha \neq j)
\]

If the variables $\alpha$, $\bar{x}$ and $\bar{y}$ do not partition the reference name space then the remaining variables are potential aliases. The reference name space is the set of all reachable references, including reference variables and references held in the store.

**Definition 8.2 (Vector Aliased)** The $(\approx_{\bar{X}}^{\bar{Y}})$ construct is generalised to permit vectors of variables. Given variable vector $\bar{\alpha}$ and vectors of variable sets $\bar{X}$ and $\bar{Y}$,

\[
(\bar{\alpha} \approx_{\bar{X}}^{\bar{Y}}) \equiv \forall l \in 1..\#\bar{\alpha} \bullet (\bar{\alpha}_{l \approx_{\bar{X}}^{\bar{Y}}})
\]

**Example 8.3 (Aliasing Annotation)** Given references $\alpha$, $\beta$, and $\gamma$ the following specification (in which $\alpha$ is aliased to $\beta$ and not aliased to $\gamma$):

\[
\alpha: [\alpha = \beta \land \beta \neq \gamma \land \gamma \uparrow \geq \beta \uparrow, \alpha \uparrow \geq \beta \uparrow]
\]

can be specified, using the aliasing annotations, as follows.

\[
\alpha: [(\alpha \approx_{\bar{X}}^{\bar{Y}}) \land \gamma \uparrow \geq \beta \uparrow, \alpha \uparrow \geq \beta \uparrow]
\]

Once all reference variables of the environment have been distributed across a variable’s alias annotations, the bottom annotation can be omitted. For example, in an environment with reference variables $\alpha$, $\beta$, and $\gamma$ of the same store, the alias construct

\[
(\alpha \approx_{\bar{X}}^{\bar{Y}})
\]

can be abbreviated as

\[\alpha = \beta\]

This signifies that $\alpha$ has an alias $\beta$ and that no other environment variable is aliased to $\alpha$. As a pragmatic constraint, the use of this syntax is not permitted when declaring fresh variables. When a reference variable is introduced, its initial value is non-deterministic and may be equal to an existing reference variable.
When a variable has no aliases it is termed a unique reference and is denoted by the following.

\[ \alpha = \emptyset \]

The following syntax, which is equivalent to True, can be used to make a lack of aliasing information explicit.

\[ (\alpha \neq \emptyset) \]

### 8.2 Reference Specifications

This section introduces a novel statement for reasoning about references, termed reference specifications. Reference specifications permit two different frame annotations that denote various constraints on the store as illustrated in Figure 8.1. By using reference specifications, the store constraints can be dealt with in a methodical manner and the specifications made more readable and consistent. The reference specification:

\[ \alpha : [pre, post]_* \]

allows the reference \( \alpha \) to be altered. This permits the assignment to \( \alpha \) of another reference, e.g., a reference variable, object field, or the result of a new command. It does not, in general, permit the store to be altered at location \( \alpha \), unless \( \alpha \) is assigned the result of a new. Since the result of a new is an unaliased location, assigning the store at that location cannot violate existing aliasing properties.

The reference specification statement:

\[ \alpha \|= [pre, post]_* \]

allows the store to be altered at the initial index \( \alpha_0 \). It does not permit the reference variable \( \alpha \) to be altered.

By combining the frame annotations, both the reference \( \alpha \) and the store at the initial index \( \alpha_0 \) may be modified.

\[ \alpha, \alpha \|= [pre, post]_* \]

---

\(^3\)Using the nomenclature of Jones [Jon92].
However, it does not in general permit the alteration of the store at the final index (α) as this would permit the alteration of the store at all indices. For example, consider the following (classical) refinement where α is assigned some intermediate value before being assigned its desired, final value.

\[
\alpha : [\alpha = 10] \subseteq \alpha := 5; \; \alpha := 10
\]

If it were permitted to arbitrarily alter the store at the final index, then a specification such as the one following could alter the entire store\(^4\) by assigning α to intermediate values and updating the store at those locations. The following specification has a precondition in which the reference α is unique and a postcondition which constrains α to be aliased at least to γ and constrains the dereference of α to be ten: α\(\uparrow\) = 10. If α\(\uparrow\) could be modified, then the following code could result.

\[
\alpha, \alpha\uparrow : [\alpha = \emptyset, (\alpha = \emptyset) \land \alpha\uparrow = 10], \alpha \subseteq \alpha := \delta; \; \alpha := \gamma; \; \alpha\uparrow := 10
\]

The alteration of the store at indices γ and δ was not intended by the original specification and may violate established properties. In contrast, modifying the store at new, un-aliased locations cannot violate an established property. To alter the store at the final index α, the frame should include the intended alias (γ). Using this approach the following code can be developed:

\[
\alpha, \gamma\uparrow : [\alpha = \emptyset, (\alpha = \emptyset) \land \alpha\uparrow = 10], \alpha \subseteq \alpha := \gamma; \; \gamma\uparrow := 10
\]

As a pragmatic concern, it is considered poor practice to alter the store through the dereference of the final value of α as it is the inclusion of γ\(\uparrow\) in the frame that permits the alteration of the store at this location. Consequently, although technically the following specification is well formed, by convention it is avoided.

\[
\alpha, \gamma\uparrow : [\alpha = \emptyset, (\alpha = \emptyset) \land \alpha\uparrow = 10]
\]

Given the two different forms of frame annotations, it is possible to formally define reference specifications.

**Definition 8.4 (Reference Specification)** Given (possibly overlapping) variable vectors (sequences of variables) \(\vec{\alpha}\) and \(\vec{\beta}\) then

\[
\vec{\alpha}, \vec{\beta}\uparrow : [\text{pre}, \text{post}]_* \equiv \text{store}, \vec{\alpha} : [\text{pre}, \text{post} \land \vec{\beta}_0 \subseteq \text{store}]
\]

where \(\vec{\beta}_0 \subseteq \text{store}\) denotes that the store remains constant except at the indices \(\vec{\beta}_0\).

\(\diamondsuit\)

The definition constrains the initial indices \(\vec{\beta}_0\) (rather than \(\vec{\beta}\)) in case the vector \(\vec{\beta}\) overlaps with \(\vec{\alpha}\).

\(^4\)Depending upon the rules for composition.
Given store $S$ and a set of reference indices $Z$, then $Z \preceq S$ is a relation denoting that the store $S$ is the same as its initial value $S_0$ except possibly at the indices in $Z$.

$$Z \preceq S \triangleq \text{First attempt}$$

$$Z \ll S = Z \ll S_0$$

This is generalised to allow unconstrained additions to store $S$ and object-refinement of the objects.

**Definition 8.5 (Constrained)**

$$Z \preceq S \triangleq (Z \ll (\text{dom} \ S_0 \ll S) \gg Z \ll S_0)$$

Figure 8.2 illustrates the additional locations that the generalisation permits alterations to.

Refinement rules analogous to the classical rules can be provided for reference specifications. For instance, the following theorem is analogous to the classical Sequential Composition (B.38) refinement rule.

**Theorem 8.6 (Reference Specification Sequential Composition)**  

Proof on page 237

In an environment with variables $\bar{x}$, with predicate $\text{mid}$ containing no initial variables except $\bar{a}_0$ (and implicitly $\text{store}_0$), and disjoint variable vectors $\bar{\alpha}$ and $\bar{\beta}$, then

$$\bar{\alpha}, \bar{\beta} \triangleright [\pre, \post]_*$$

$$\models [\text{con } \bar{A}, \text{STORE } \bullet$$

$$\bar{\alpha}, \bar{\beta} \triangleright [\pre, \text{mid}]_* ;$$

$$\bar{\alpha}, \bar{\beta} \triangleright (\text{mid})[\bar{a}_0, \text{store}_0 \backslash \bar{A}, \text{STORE}]$$

$$\models (\text{post})[\bar{a}_0, \text{store}_0 \backslash \bar{A}, \text{STORE}]_*. $$
The following theorem permits the introduction of a reference clone command.

**Theorem 8.7 (Introduce Reference Clone Command)**  
Proof on page 238  
For expression $\beta$,  
$$\alpha : [\alpha = \varnothing \land \alpha \-models^\subseteq \beta_0] \subseteq \alpha := \text{new } \beta$$  
where $\beta_0$ is $\beta[\text{store}, \alpha\-\text{store}_0, \alpha_0]$.

Reference cloning for field updates can be defined in an analogous manner to reference cloning for variables (Definition 7.26).

**Definition 8.8 (Reference Clone for Field Updates)**  
For reference $o$ and expression $\beta$,  
$$o \uparrow^t \alpha := \text{new } \beta$$  
$$\models \left[ \text{var } t : \text{Ref} \bullet t : \left[ (\forall i \bullet t \notin \text{dom store}_i) \right] ; t \uparrow_{\text{store}_j} := \beta ; o \uparrow^t \alpha := t \right]$$

The following theorem permits the introduction of a reference clone for field updates.

**Theorem 8.9 (Introduce Reference Clone Field Update)**  
Proof on page 241  
For expression $\beta$,  
$$o \uparrow^t \alpha : \left[ o \uparrow^t \alpha = \varnothing \land o \uparrow^t \alpha \models^\subseteq \beta_0 \right] \subseteq o \uparrow^t \alpha := \text{new } \beta$$  
where $\beta_0$ is $\beta[\text{store}, o\-\text{store}_0, o_0]$.

The following theorem illustrates the effect of removing dereferenced variables from the frame of a reference specification.

**Theorem 8.10 (Dereference Contract Frame)**  
Proof on page 242  
For disjoint $\alpha$ and $\beta$,  
$$\overline{\alpha} \uparrow, \overline{\beta} \uparrow, \overline{\gamma} : \left[ \text{pre } , \text{post } \Lambda \overline{\alpha} \models^\subseteq \overline{\alpha} \right] \subseteq \overline{\beta} \uparrow, \overline{\gamma} : \left[ \text{pre } , \text{post } \right]$$

### 8.3 Data refinement via Inverse Commands

In preparation for subsequent sections, a technique related to one discussed by Back and von Wright [BvW90] is introduced. This technique allows the development of a program through a temporary abstraction to a universal join homomorphic environment\(^5\). By temporarily abstracting the environment, the more mundane aspects of the program can be reasoned about using a simpler semantics.

---

\(^5\)Desideratum 2.6 defines universal join homomorphisms.
The technique involves two data refinements where the second cancels or inverts the first. Consequently, the specification and implementation variables of the first data refinement are swapped to form the implementation and specification variables of the second. In special cases, data refinement via inverse commands maintains a form of monotonicity that allows a code segment to be developed in isolation and later incorporated in lieu of the original code. To prevent later terminology confusion, the original environment is termed the *initial* and the latter, abstracted environment is termed the *mirror environment*.

To apply this technique, a data refinement command, \( \text{rep} \), is chosen to allow the data refinement from the initial environment to the mirror environment. Additionally, another command, \( \text{rep}' \), is chosen to perform a data refinement in the reverse direction: from the mirror environment to the initial environment. To ensure that the effects of \( \text{rep}' \) invert those of \( \text{rep} \), the commands \( \text{rep} \) and \( \text{rep}' \) are chosen so that:

\[
\text{rep}; \text{rep}' \sqsubseteq \text{skip} \tag{8.1}
\]

and

\[
\text{skip} \sqsubseteq \text{rep}; \text{rep}' \tag{8.2}
\]

To allow the classical data refinement rules to be applied, both \( \text{rep} \) and \( \text{rep}' \) are also constrained to be monotonic

\[
(\forall s \bullet \phi \Rightarrow \gamma) \Rightarrow (\forall i \bullet \text{rep} \phi \Rightarrow \text{rep} \gamma)
\]

and \( \lor \)-distributive (also known as universal-join-homomorphic).

\[
\text{rep} (\lor_i \phi_i) = \lor_i (\text{rep} \phi_i)
\]

Consequently, given a program segment \( P \), the following development occurs:

\[
P \equiv \begin{align*}
\text{skip}; P \\
\text{Equation (8.2)}
\end{align*} \\
\text{rep}; \text{rep}'; P
\]

Using classical data refinement techniques, \( P \) is data refined under \( \text{rep}' \) to give \( P' \) where \( \text{rep}'; P \sqsubseteq P'; \text{rep}' \).

\[
\text{Definition of Data Refinement (2.3)} \\
\text{rep}'; P'; \text{rep}'
\]

Since \( \text{rep}' \) maps the initial environment to a more abstract mirror environment, \( P' \) is a program segment which is more easily reasoned about than the analogous \( P \) program segment. After development of \( P' \), a refinement \( Q' \) is obtained, i.e., \( P' \sqsubseteq Q' \).

\[
\text{Monotonicity of refinement.} \\
\text{rep}; Q'; \text{rep}'
\]
Finally, $Q'$ is transformed back to the initial environment by data refining it under $rep$ giving $Q''$.

\[ Q'' \]

\[ rep; \ rep' \]

\[ Q'' \]

**Definition of Data Refinement (2.3)**

$Q''$; $rep$; $rep'$

\[ Equation \ (8.1) \]

$Q''$

In summary, $P \subseteq Q''$ if $P \subseteq P'$, $P' \subseteq Q'$, $Q' \subseteq Q''$, and $rep'$ is the inverse of $rep$. Examples of the use of this technique are provided in Sections 8.4 and 8.5. This technique differs from that of Back and von Wright in that their final data refinement ($Q' \subseteq rep \ Q''$) is universal-meet-homomorphic, rather than an inverse. Universal-meet-homomorphic is also known as $\wedge$-distributive:

\[ rep \ (\wedge_\phi \phi_i) = \wedge_\phi \ (rep \ \phi_i) \]

### 8.4 Coalesced Programs

As identified earlier, one form of aliasing difficulty occurs when the alteration of a variable violates a property for an alias. A more detailed example of how a property may be violated through aliasing, and naive reasoning, is now presented. This example is used to motivate and describe a novel technique for avoiding such violations. Consider the following refinement, starting with a reference specification which permits the modification of the store at indices $\alpha$ and $\beta$ to establish the postcondition under the assumption that $\alpha$ and $\beta$ are aliased.

\[ \alpha \vdash_\beta \vdash \left[ \alpha = \beta, \ \alpha \leq 9 \land \beta \geq 4 \right] \]

**Reference Specification Sequential Composition (8.6)**

\[ \alpha \vdash_\beta \vdash \left[ \alpha = \beta, \ \alpha \leq 9 \land \beta \geq 4 \right] ; \ \alpha \vdash_\beta \vdash \left[ \alpha = \beta \land \beta \geq 4, \ \alpha \leq 9 \land \beta \geq 4 \right] \]

According to classical techniques, it would now seem desirable to use a strengthen postcondition rule similar to Morgan’s to remove $\beta \geq 4$ from the second postcondition given that it is already satisfied by the precondition. Such a rule does not exist. If one did, then the following would be a refinement (after a further weaken precondition).

\[ \subseteq \alpha \vdash_\beta \vdash \left[ \alpha = \beta, \ \alpha = \beta \land \alpha \geq 4 \right] ; \ \alpha \vdash_\beta \vdash \left[ \alpha = \beta, \ \alpha \leq 9 \right] \]

Using the knowledge of the alias of $\alpha$ with $\beta$, this refines to:

\[ \subseteq \alpha \vdash_\beta \vdash \left[ \alpha = \beta, \ \alpha = \beta \land \alpha \geq 4 \right] ; \ \alpha \vdash_\beta \vdash \left[ \alpha = \beta, \ \alpha \leq 9 \right] \]

This could then be refined to:

\[ \subseteq \alpha \vdash := 10; \ \alpha \vdash := 3 \]
This program violates the conjunct \( \beta \geq 4 \) of the original postcondition. If however, the original specification were given as:

\[
\alpha \models [ \alpha = \beta \land \alpha \leq 9 \land \alpha \geq 4 ]
\]

then the strengthen postcondition rule could not be applied to weaken the postcondition from \( \alpha \leq 9 \land \alpha \geq 4 \) to \( \alpha \leq 9 \). The systematic replacement of all occurrences of \( \beta \) with its alias \( \alpha \) is a solution which would avoid the violation of properties due to aliasing. This observation motivates an original technique termed coalescing. Coalescing is the unification of references, in this case \( \alpha \) and \( \beta \). The technique is an instantiation of the data refinement technique described in Section 8.3. Consequently it involves three steps starting with the transformation of the program to an environment where the two references are coalesced, leaving only one reference in the environment. The program is then refined and finally the resulting code is transformed back to the original environment in which the second reference exists. The final transformation typically requires modifications such as the inclusion of additional assignment statements. Both transformation processes, though, are syntactic, trivial and require no additional proof.

### 8.4.1 Transforming to a Coalesced Program

Transforming to a coalesced environment involves data refining a program segment so that two or more previously separate reference variables are unified and hence accessible via a single variable. The following rules are used for coalescing specifications.

Although the coalescing of a specification is a data refinement, the operator ‘coalesces-to’ is used here to indicate that unlike other data refinements, the development of the (sub)program is not complete until the latter, third, step of transforming back to the original environment is performed.

For all rules in this section, the variables \( \bar{\beta} \) are coalesced to the variable \( \alpha \). The variable \( \alpha \) is termed the primary. Coalescing \( \bar{\beta} \) with \( \alpha \) requires the replacement of each element of the vectors \( \bar{\beta} \) and \( \bar{\beta}_0 \) with \( \alpha \) and \( \alpha_0 \) respectively.

**Theorem 8.11 (Unannotated Coalesced Specification)**  

Proof on page 243  

For disjoint variable sets \( \bar{g}, \bar{h}, \{ \alpha \} \) and \( \bar{\beta} \):

\[
\alpha, \bar{\beta}, \bar{g}, \bar{h} : \left[ pre \land \alpha = \bar{\beta}, \text{ post} \right]
\]

coalesces-to

\[
\alpha, \bar{g}, \bar{h} : \left[ \text{pre}[^{\bar{\beta}} \Delta \alpha, ..., \alpha] \land \alpha = \varnothing, \text{ post}[^{\bar{\beta}} \Delta \alpha_0, ..., \alpha, \alpha_0, ..., \alpha_0] \right]
\]

The only restriction on the frame annotations of \( \alpha \) and \( \bar{\beta} \) is that they are the same, i.e., the frame is either \( \alpha, \bar{\beta}, \bar{g}, \bar{h} \), or \( \alpha^\uparrow, \bar{\beta}^\uparrow, \bar{g}, \bar{h} \) or \( \alpha, \alpha^\uparrow, \bar{\beta}, \bar{\beta}^\uparrow, \bar{g}, \bar{h} \). Thus there are two other similar rules:
Chapter 8. Towards a Reference Semantics

Theorem 8.12 (Dereferenced Coalesced Specification)  Proof on page 244  For disjoint variable sets $\tilde{g}, \tilde{h}, \{\alpha\}$ and $\beta$:

$$\alpha', \beta', \tilde{g}, \tilde{h} : \left[ \text{pre } \land \alpha = \beta, \ post \right]_*$$

coalesses-to

$$\alpha', \tilde{g}, \tilde{h} : \left[ \text{pre}[\tilde{\beta}\setminus\alpha, ..., \alpha] \land \alpha = \emptyset, \ post[\tilde{\beta}, \tilde{\beta}_0 \setminus \alpha, ..., \alpha, \alpha_0, ..., \alpha_0] \right]_*$$

Example 8.13 (Transforming to a Coalesced Specification)  For example, the reference specification

$$\alpha', \beta : \left[ \alpha = \beta, \ \alpha \leq 9 \land \beta \geq 4 \right]_*$$

coalesses-to

$$\alpha' : \left[ \alpha = \emptyset, \ \alpha \leq 9 \land \alpha \geq 4 \right]_*$$

\diamond

Theorem 8.14 (Dual Coalesced Specification)  Proof on page 245  For disjoint variable sets $\tilde{g}, \tilde{h}, \{\alpha\}$ and $\beta$:

$$\alpha, \alpha', \tilde{\beta}, \tilde{\beta}', \tilde{g}, \tilde{h} : \left[ \text{pre } \land \alpha = \beta, \ post \right]_*$$

coalesses-to

$$\alpha, \alpha', \tilde{g}, \tilde{h} : \left[ \text{pre}[\tilde{\beta}\setminus\alpha, ..., \alpha] \land \alpha = \emptyset, \ post[\tilde{\beta}, \tilde{\beta}_0 \setminus \alpha, ..., \alpha, \alpha_0, ..., \alpha_0] \right]_*$$

8.4.2 Transforming from a Coalesced Program

After refinement of the coalesced program, the resulting program is transformed back into the environment containing the coalesced variable. The transformation is a piecewise process, whereby each construct of the program is transformed using the rule appropriate for that language construct. The rules are syntactic, the process is straightforward and no additional proof is required. The operator ‘uncoalesces-to’ is used here as a complement to the ‘coalesces-to’ operator.

The rules for specifications and assignments are given. For most other language constructs the transformation is an identity function and hence leaves the program syntactically the same. For instance, for assignments to variables other than the primary, the transformation is the identity function. For assignments to the primary an additional assignment is inserted to re-synchronise the coalesced variables to the primary.
Theorem 8.15 (Assignments to the Primary)  Proof on page 246  For assignments to primary variable \( a \) the following transformation is applicable.

\[
\alpha := X;
\]

uncoalesces-to

\[
\alpha := X; \; \beta := \alpha, \ldots, \alpha
\]

Dereferenced assignments to the primary, e.g., \( \alpha \triangleright:= X \), remain the same as they actually alter the store, not the reference (\( \alpha \)).

For specifications, the precondition can be optionally strengthened with the aliasing of \( \alpha \) to \( \beta \), i.e., \( \alpha=\beta \). The postcondition is strengthened with \( \alpha=\beta \) as the code following the specification relies on the aliasing. There are three rules provided for the transformation from a coalesced to an uncoalesced specification; one for each frame element annotation: none, \( \dagger \) and one for both.

Theorem 8.16 (Unannotated Uncoalesced Specification)  Proof on page 246  When uncoalescing a specification, the coalesced variables (\( \beta \)) are returned to the frame and the postcondition is strengthened to ensure that subsequent code can rely on the aliasing of \( \alpha \) with \( \beta \).

\[
\alpha; \; \text{pre} \land \alpha=\emptyset, \; \text{post}\]

uncoalesces-to

\[
\alpha, \beta; \; [ \alpha=\beta \land \text{pre} , \; \alpha=\beta \land \text{post} ]
\]

Theorem 8.17 (Dereferenced Uncoalesced Specification)  Proof on page 247  For specifications involving a frame with a dereferenced primary:

\[
\alpha\dagger; \; \text{pre} \land \alpha=\emptyset, \; \text{post}\]

uncoalesces-to

\[
\alpha\dagger, \beta\dagger; \; [ \alpha=\beta \land \text{pre} , \; \alpha=\beta \land \text{post} ]
\]

The strengthening of the postcondition is not required as the frame annotation of \( \alpha \) prevents the reference \( \alpha \) from being altered.

Theorem 8.18 (Dual Uncoalesced Specification)  Proof on page 249  For disjoint variable sets \( \tilde{g}, \tilde{h}, \{\alpha\} \) and \( \tilde{\beta} \):

\[
\alpha, \alpha\dagger, \tilde{g}, \tilde{h}; \; [ \text{pre} \land \alpha=\emptyset , \; \text{post} ]
\]

uncoalesces-to

\[
\alpha, \alpha\dagger, \beta\dagger, \tilde{g}, \tilde{h}; \; [ \alpha=\beta \land \text{pre} , \; \alpha=\beta \land \text{post} ]
\]
Example 8.19 (Coalescing) Starting with a reference specification, the coalescing of \( \alpha \) and \( \beta \) is used here to refine in the absence of aliasing.

\[
\alpha, \beta : [\alpha^{=} \land \gamma{=} 5 \land \alpha{[} \leq 9 \land \beta{[} \geq 4 ]{.}
\]

coalesces-to, using Theorem Unannotated Coalesced Specification (8.11),

\[
\alpha : [\gamma{=} 5 \land \alpha{=} \sigma \land \alpha{[} \leq 9 \land \alpha{[} \geq 4 ]{.}
\]

This can be refined to the following assignment which aliases \( \gamma \) to \( \alpha \):

\[
\varnothing \alpha := \gamma
\]

After transformation back to the original environment using Theorem Assignments to the Primary (8.15), the following code results:

\[
\alpha := \gamma; \beta := \alpha
\]

Consequently, it has been shown that

\[
\alpha, \beta : [\alpha^{=} \land \gamma{=} 5 \land \alpha{[} \leq 9 \land \beta{[} \geq 4 ]{,} \varnothing \alpha := \gamma; \beta := \alpha
\]

Alternatively, this code could have been produced via the following (abbreviated) development:

\[
\alpha, \beta : [\alpha^{=} \land \gamma{=} 5 \land \alpha{[} \leq 9 \land \beta{[} \geq 4 ]{,}
\]

\[
\varnothing \alpha := \gamma; \beta := \alpha
\]

As an aside, notice that the original specification does not require the aliasing of \( \alpha \) with \( \beta \) in the postcondition. This constraint was introduced by coalescing the specification.

8.4.3 Soundness of Coalesced Programming

The development of programs using coalescing is achieved by temporarily transferring the program to a different environment using the data refinement by inverse commands technique introduced in Section 8.3. The initial environment variables are the variables that are coalesced, i.e., \( \bar{\beta} \). These are removed and no variables are introduced. That is, there are no mirror environment variables. Using the abstraction invariant

\[
AI \triangleq \bar{\beta} = \alpha
\]

the data refinement command for transforming to a coalesced environment is:

\[
rep \ p \triangleq (\exists \bar{\beta} \bullet p \land \bar{\beta} = \alpha)
\]

The data refinement command for transforming from a coalesced environment is:

\[
rep' \ p \triangleq p \land \bar{\beta} = \alpha
\]
Theorem 8.20 (Inversed reps)  Proof on page 249  Showing that coalesced programming is an instantiation of the ‘data refinement via inverse commands’ technique requires proving that \( rep' \) is an inverse of \( rep \) as shown in Section 8.3. That is,

\[
\text{skip} \equiv \text{rep; rep'}
\]

and hence, as a corollary:

\[
\text{skip} \sqsubseteq \text{rep; rep'} \land \text{rep'} \sqsubseteq \text{skip}
\]

The data refinement rules provided in Sections 8.4.1 and 8.4.2 use \( rep \) and \( rep' \). For the transformation to coalescent programs it is intended that only specifications are coalesced\(^6\). For this purpose, rules 8.11, 8.12 and 8.14, have been provided. Rules must also be provided for all possible language constructs for the transformation back to the initial environment. For assignments and specifications, alterations are typically required, and hence rules 8.15, 8.16, 8.17 and 8.18 are provided. Most constructs, however, are transformed under the identity function; and as such most of these rules are omitted. One such rule, 8.21, is included for instructional purposes: assignments to variables other than the primary variable are transformed to themselves when they are uncoalesced.

Theorem 8.21 (Assignments to Non-Primaries)  Proof on page 250  Given a variable \( \gamma \), which is not the primary variable (i.e., \( \alpha \)),

\[
\gamma := X
\]

uncoalesces-to

\[
\gamma := X
\]

8.5 Semantics Conversion

This section presents an innovative technique termed semantics conversion. Semantics conversion involves temporarily data refining reference variables to value variables. This allows simple reasoning using a semantics for values rather than the complex, array-like reasoning needed for a semantics for references.

A reference semantics code segment is (mechanically) ‘abstracted’ to a value semantics program by replacing the variable dereferences with analogous value variables. The program is then refined using classical refinement rules (e.g., those of Morgan). The resulting program is then (mechanically) transformed back to a semantics for references. Like coalescing, the soundness of semantics conversion is achieved using the proof technique described in Section 8.3. Section 8.5.1 provides the data refinement rules for transforming to a value semantics, Section 8.5.2 provides the data refinement rules for transforming to a reference semantics, and Section 8.5.3 presents the soundness proof.

\(^6\)There is no technical reason why other language constructs could not be coalesced.
8.5.1 Transforming to a Value Semantics

This section presents rules for the transformation of reference specifications to value semantics specifications. The rules assume that the variables $\bar{\alpha}$ and $\bar{\beta}$ are being converted from reference to value semantics variables.

**Theorem 8.22 (Transforming Reference Specification)** Proof on page 251 Translating a reference specification with unaliased variables $\bar{\alpha}$ and $\bar{\beta}$ involves replacing all dereferences of $\bar{\alpha}$ and $\bar{\beta}$ with direct variable accesses. Given variables $\bar{\alpha} : \text{Ref} \{\bar{\Delta}\}$ and $\bar{\beta} : \text{Ref} \{\bar{\Xi}\}$, the following specification

$$\bar{\alpha} : [\alpha = , \beta = \text{pre}[\bar{\alpha}, \bar{\alpha}_0, \bar{\beta}, \bar{\beta}_0 \\backslash \bar{\alpha}_1, \bar{\beta}_1, \bar{\beta}_0], \text{post}[\bar{\alpha}, \bar{\alpha}_0, \bar{\beta}, \bar{\beta}_0 \\backslash \bar{\alpha}_1, \bar{\beta}_1, \bar{\beta}_0]]$$

encodes-as (with respect to $\bar{\alpha}$ and $\bar{\beta}$)

$$\bar{\alpha} : [\text{pre} , \text{post}]$$

where $\bar{\alpha} : \bar{\Delta}$ and $\bar{\beta} : \bar{\Xi}$.

The operator ‘encodes-as’ is used to indicate that semantics conversion is a two-phase process. The process involves the implicit removal of reference variables $\bar{\alpha}$ and the implicit addition of value variables $\bar{\alpha}$.

**Example 8.23 (Transforming Reference Specification)** The variables in the reference specification

$$\alpha : [\alpha = \beta = \text{pre}[\bar{\alpha}, \bar{\alpha}_0, \bar{\beta}, \bar{\beta}_0 \\backslash \bar{\alpha}_1, \bar{\beta}_1, \bar{\beta}_0], \text{post}[\bar{\alpha}, \bar{\alpha}_0, \bar{\beta}, \bar{\beta}_0 \\backslash \bar{\alpha}_1, \bar{\beta}_1, \bar{\beta}_0]]$$

have no aliases. By converting all references ($\alpha$ and $\beta$) to value variables, the specification can be transformed to:

$$\alpha : [\alpha = \beta , \alpha = \beta + 1]$$

The refinement of this specification is significantly easier. For example, using Simple Specification (B.52) it refines to

$$\alpha := \beta + 1$$

or alternatively to

$$\alpha := \alpha + 1$$

This code, once transformed back to the reference semantics, is a refinement of the original reference specification (see Example 8.25 below).
8.5.2 Transforming to a Reference Semantics

This section provides the rules for transforming from a value semantics to a reference semantics. Although all language constructs must be provided with a transformation rule for the technique to be complete, only several indicative rules are provided here. The rules assume that the variables \( z \) are being converted from value to reference semantics variables.

**Theorem 8.24 (Transforming Value Semantics Assignments)**  Proof on page 258
For the transformation of value semantics variables \( z \) to reference semantics variables, the assignment

\[
D := E
\]

decodes-as (with respect to \( z \))

\[
(D := E)[z]_{z}
\]

The operator ‘decodes-as’ is used to indicate rules for the second phase of the semantics conversion process.

**Example 8.25 (Transforming Value Semantics Assignments)**  For example, the conversion of (integer) variables \( \alpha \) and \( \beta \) in

\[
\alpha := \beta + 1
\]

will result in the reference semantics program:

\[
\alpha[1] := \beta[1] + 1
\]

\( \diamond \)

**Theorem 8.26 (Transforming Value Semantics Specification)**  Proof on page 258
Transforming a value semantics specification to a reference specification involves replacing direct variable access with dereferenced variables. For the transformation of value semantics variables \( z : \bar{\Delta} \) (and subset \( \bar{\alpha} \), i.e., \( \bar{\alpha} \subseteq \bar{\Delta} \)) to reference semantics variables, the following specification

\[
\bar{\alpha} : [\text{pre} , \text{post}]
\]

decodes-as (with respect to \( \bar{\Delta} \))

\[
\bar{\alpha}, \bar{\alpha}[1] : [\text{pre}[z,\bar{\Delta}][1] \wedge \bar{\Delta}^{\alpha} , \bar{\Delta}^{\bar{\alpha}} \wedge \text{post}[\bar{\alpha_0}, \bar{\alpha}_0[1_0], \bar{\Delta}[1]]]_1
\]

in an environment where \( \bar{\Delta} : \text{Ref} \{ \bar{\Delta} \} \).
8.5.3 Proof of Semantic Conversions

A proof is given in this section, showing that the technique and rules presented in the previous section are sound. This soundness is shown using the technique presented in Section 8.3.

The rules of Sections 8.5.1 and 8.5.2 are data refinements using an initial environment with \(\text{store}\) and reference variables \(\bar{z}\) and mirror environment with value variables \(\bar{z}'\) and \(\text{store}'\). They use the following abstraction invariant which equates the dereferences of \(\bar{z}\) with the value variables \(\bar{z}'\) and removes \(\bar{z}\) from the domain of \(\text{store}\) to give \(\text{store}'\).

\[
AI \triangleq \bar{z} = \bar{z}' \land \bar{z} \not\in \text{store} = \text{store}'
\]

The predicate transformer\(^7\) for converting to a value semantics is:

\[
\text{rep} \ p \triangleq (\exists \bar{z}, \text{store} \bullet p \land AI)
\]

The following data type invariant is also used to ensure that the references remain unaliased when converting to a semantics for references:

\[
\text{DTI} \triangleq \bar{z} = \emptyset
\]

The predicate transformer for converting to a semantics for references is:

\[
\text{rep}' \ p \triangleq (\exists \bar{z}', \text{store}' \bullet p \land AI \land \text{DTI})
\]

Soundness relies on showing that \(\text{rep}; \text{rep}'\) refines to \text{skip} and vice versa. The following two theorems fulfill this requirement.

**Theorem 8.27 (Inversed reps for Semantic Conversion)** Proof on page 256

\[
\text{skip} \subseteq \text{rep}; \text{rep}'
\]

**Theorem 8.28 (Inversed reps for Semantic Conversion B)** Proof on page 257

\[
\text{rep}; \text{rep}' \subseteq \text{skip}
\]

8.5.4 Simulation

Since the rules presented in Section 8.5.2 are data refinement rules, they can also be used for the data refinement of local variable blocks. Consequently, a value variable block can be simulated, via data refinement, by a reference variable block.

\(^7\)Predicate transformers and commands are used interchangeably.
CHAPTER 8. TOWARDS A REFERENCE SEMANTICS

Theorem 8.29 (Simulating a Value Semantics) Given semantics for values program $M$ and semantics for references program $O$,

$$\left\lfloor \text{var } \bar{v} : \tau(\bar{V}) \leftarrow \bar{V} \cdot M \right\rfloor$$

$$\left\lfloor \text{var } \text{store } : \text{Ref} \{\tau(\bar{V})\} \leftarrow \tau(\bar{V}) \cdot \right\rfloor$$

$$\left\lfloor \text{var } \bar{v} : \text{Ref} \{\tau(\bar{V})\} \leftarrow \text{new } \bar{V} \cdot O \right\rfloor$$

provided $M$ decodes-as $O$ (with respect to $\bar{v}$).

Proof

To aid readability, value variables are denoted by $\bar{v}'$ and reference variables by $\bar{v}$.

The rules of Section 8.5.2 assume that a store is in the environment. Consequently, to use those rules, a store must be added to the program.

$$\left\lfloor \text{var } \bar{v}' : \tau(\bar{V}) \leftarrow \bar{V} \cdot M \right\rfloor$$

Introduce Local Variable Block (B.55)

$$\left\lfloor \text{var } \text{store } : \text{Ref} \{\tau(\bar{V})\} \leftarrow \tau(\bar{V}) \cdot \right\rfloor$$

$$\left\lfloor \text{var } \bar{v} : \tau(\bar{V}) \leftarrow \bar{V} \cdot N \right\rfloor$$

For specifications in $M$ the frame may be expanded with $\text{store}$ (as is shown in the Introduce Local Variable Block (B.55) rule) to produce analogous versions in $N$. For simplicity, other language constructs remain the same, simulating the invariance of $\text{store}$.

The data refinement command ($\text{rep}'$) for the rules of Section 8.5.2 equates the dereference of $\bar{v}$ with the value variables $\bar{v}'$, allows $\text{store}$ to differ from $\text{store}'$ only at indices $\bar{v}$, and ensures that $\bar{v}$ remains unaliased: $\text{rep}' p \triangleq (\exists \bar{v}', \text{store}' \cdot \bar{v}' \leftarrow \bar{v} \land \{\bar{v}'\} \leftarrow \text{store} = \{\bar{v}\} \leftarrow \text{store}' \land \bar{v}' \leftarrow \emptyset \land p)$. The result of data refining the variables $\bar{v}'$ using $\text{rep}'$ is:

$$\left\lfloor \text{var } \text{store } : \text{Ref} \{\tau(\bar{V})\} \leftarrow \tau(\bar{V}) \cdot \right\rfloor$$

$$\left\lfloor \text{var } \bar{v} : \text{Ref} \{\tau(\bar{V})\} \mid \bar{v}' \leftarrow \emptyset \cdot O \right\rfloor$$

where $N \preceq_{\text{rep}'} O$ (e.g., by mechanical transformation using the data refinement rules provided in Section 8.5.2). This data refinement introduces an invariant that constrains $\bar{v}$ to be unaliased ($\bar{v}' \leftarrow \emptyset$). This constraint can be established by assigning $\bar{v}$ to the result of a $\text{new}$.

$$\left\lfloor \text{var } \text{store } : \text{Ref} \{\tau(\bar{V})\} \leftarrow \tau(\bar{V}) \cdot \right\rfloor$$

$$\left\lfloor \text{var } \bar{v} : \text{Ref} \{\tau(\bar{V})\} \leftarrow \text{new } (\bar{V}) \cdot O \right\rfloor$$

In summary, the rules of Section 8.5.2 can be used to simulate a value variable block with a reference variable block.

QED
8.6 Singly Linked List Example

In this section the implementation of a stack as a singly-linked list is presented. This example is based on a data refinement presented by Bancroft [Ban97]. Two separate data refinements are used in this version. The first transforms an object with a sequence field into an object with a (value semantics) recursively encapsulated list where each node of the list contains the tail of the list through its \textit{next} field. For instance, the sequence:

\[(1, 2, 3)\]

is represented by the ‘list’:

\[
\begin{aligned}
&\text{object} \\
&\quad\text{field } \text{element} = 1 \\
&\quad\text{field } \text{next} = \left( \begin{array}{c}
\text{object} \\
&\quad\text{field } \text{element} = 2 \\
&\quad\text{field } \text{next} = \left( \begin{array}{c}
\text{object} \\
&\quad\text{field } \text{element} = 3 \\
&\quad\text{field } \text{next} = \text{null}
\end{array} \right)
\end{array} \right)
\end{aligned}
\]

The end of the ‘list’ is denoted by \textit{null}. Although the list’s representation is somewhat verbose, it captures the idea that, within a semantics for values, objects are encapsulated.

After simplification, through algorithmic refinement, the resulting object is data refined to use a reference semantics; thereby reducing the recursively encapsulated list to a classical linked list implementation. That is, rather than each object encapsulating the object that represents the list’s tail, each object contains a reference to the next element in the list.

\textbf{First Data Refinement}  

The initial specification includes a \textit{sequ} field containing a sequence of natural numbers, a \textit{push} method for prepending elements to the sequence and a \textit{pop} method for removing elements from the stack. The prefix ‘\textit{self}.’ has been dropped for clarity.

\[
\begin{aligned}
&\text{object} \\
&\quad\text{private field } \text{sequ} : \text{seq}\mathbb{N} := \langle \rangle \\
&\quad\text{method } \text{push(value } f : \mathbb{N} ) \equiv \text{sequ}:: \left[ \text{sequ} = \langle f \rangle \land \text{sequ}_0 \right] \\
&\quad\text{method } \text{pop(result } f : \mathbb{N} ) \equiv \text{sequ}, f:: \left[ \begin{array}{c}
\text{sequ} \neq \langle \rangle \\
\text{sequ}_0 = \langle f \rangle \land \text{sequ}
\end{array} \right]
\end{aligned}
\]

This specification is data refined to a linked list that uses recursively encapsulated node (\textit{Node}) objects. Each node of the linked list has three fields; an \textit{isNull} field which determines if the node represents \textit{null}, a \textit{next} field holding the tail of the linked list and an
element field containing the node’s data.

Node ≜ Object
  field isNull : Bool
  field element : Nat
  field next : Node
end

The object null is arbitrarily chosen from the set of objects of type Node where the isNull field is set to true. Given the implementation variable head which denotes the head node of the linked list, the data refinement is performed under the following abstraction function.

\[ af(n : Node) ≜ \text{if } n \text{.isNull then } \langle \rangle \text{ else } (n \text{.element}) \triangleright af(n \text{.next}) \]

This function maps null Nodes to the empty sequence and proper Nodes to the sequence formed by concatenating the Node’s element with the sequence returned from the abstraction function applied to the next field.

Using the abstraction invariant

\[ AI ≜ \text{seq} = af(head) \]

the specification is data refined to the following object, replacing the specification variable \( \text{seq} \) with the implementation variable \( \text{head} \).

\[ \leq^c_{\text{rep}} \]

object

private field head : Node := null

method push(value f : Nat) ≜
  head:: \[ af(head) = \langle f \rangle \triangleright af(head_0) \]

method pop(result f : Nat) ≜
  head,f:: \[ af(head) \neq \langle \rangle , \text{ af(head}_0) = \langle f \rangle \triangleright af(head) \]
end

where \( rep q \triangleq (\exists self \bullet q \land AI) \). This data refinement can then be algorithmically refined further. For instance, the push operation:

[\[ af(head) = \langle f \rangle \triangleright af(head_0) \]\]

refines as follows. Since \( af(head) = \langle f \rangle \triangleright af(head_0) \), it is known that \( af(head) \neq \langle \rangle \) and hence \( \neg head \text{.isNull} \), i.e., head is not null. From the definition of \( af \), for \( \neg head \text{.isNull} \),

\[ af(head) = \langle head \text{.element} \rangle \triangleright af(head \text{.next}) \]

Hence the push operation is equivalent to:

\[ head:: \[ \langle head \text{.element} \rangle \triangleright af(head \text{.next}) = \langle f \rangle \triangleright af(head_0) \land \neg head \text{.isNull} \]
The postcondition can be strengthened by equating \( f \) with \( \text{head.element} \) and \( \text{head}_0 \) to \( \text{head.next} \):

\[
\begin{align*}
\text{head.element} &= f \land \text{head.next} \sqsubseteq \text{head}_0 \\
\equiv & \quad \text{head.element} = f \land \text{head.next} \sqsubseteq \text{head}_0 \land \langle f \rangle \prec \alpha f(\text{head}_0) = \langle f \rangle \prec \alpha f(\text{head}_0) \\
\Rightarrow & \quad \text{Object-Refinement Monotonic Predicate (7.8)} \\
\langle \text{head.element} \rangle \prec \alpha f(\text{head.next}) = \langle f \rangle \prec \alpha f(\text{head}_0)
\end{align*}
\]

Consequently Specification (8.3) refines to:

\[
\text{head}:: [\text{head.element} = f \land \text{head.next} \sqsubseteq \text{head}_0 \land \neg \text{head.isNull}] \quad (8.4)
\]

\( \equiv \) Introduce Field Update (Semantics for Values) (7.29)

\[
\text{head} := \text{object}
\begin{align*}
\text{field isNull} &:= \text{false} \\
\text{field next} &:= \text{head} \\
\text{field element} &:= f
\end{align*}
\]

Since a value semantics is in use, the assignment of \( \text{head} \) to the \( \text{next} \) field of \( \text{head} \) actually copies the entire list into the \( \text{next} \) field (not just a reference).

An analogous refinement may be performed for the \( \text{pop} \) operation thereby producing the following, semantics for values object:

\[
\text{object}
\begin{align*}
\text{private field } \text{head} &:= \text{Node} := \text{null} \\
\text{method } \text{push} (\text{value} f : \text{N}) &\equiv \\
\text{head} &:= \text{object} \\
\text{isNull} &:= \text{false} \\
\text{element} &:= f \\
\text{next} &:= \text{head}
\end{align*}
\]

\[
\text{method } \text{pop} (\text{result} f : \text{N}) \equiv \\
\{ \neg \text{head.isNull} \} \\
f := \text{head.element}; \\
\text{head} := \text{head.next}
\]

\textbf{Second Data Refinement} Subsequently, a program with a reference semantics can be developed. For the \( \text{push} \) operation, the following code is developed.

\[
\text{head} := \text{new } (\text{object isNull = false, element = f, next = head end})
\]

where \( \text{head} \) is of the type \( \text{Ref } \{ \text{Node'} \} \) and

\[
\text{Node'} \equiv \text{Object}
\begin{align*}
\text{field isNull} &:= \mathbb{B} \\
\text{field next} &:= \text{Ref } \{ \text{Node'} \} \\
\text{field element} &:= \text{N}
\end{align*}
\]
Applying Theorem 8.26 to Specification (8.4), transforming value variables head and next to references, produces the following specification.

$$\text{head, head}':: \begin{cases} \text{head} = \emptyset, & \text{head}' \cdot \text{element} = f \land \text{head}' \cdot \text{next}' \sqsubseteq \text{head}_0 \uparrow_0 \land \\ \neg \text{head}' \cdot \text{isNull} \land \text{head} = \emptyset \end{cases}$$

- Object Specification Strengthen Postcondition (7.19)

$$\text{head, head}':: \begin{cases} \text{head} = \emptyset, & \text{head}' \cdot \text{element} = f \land \text{head}' \cdot \text{next}' \sqsubseteq \text{head}_0 \uparrow_0 \land \\ \neg \text{head}' \cdot \text{isNull} \land \text{head} = \emptyset \land \text{head}_0 \uparrow_0 \sqsupseteq \text{head}_0 \uparrow_0 \end{cases}$$

- Dereference Contract Frame (8.10)

$$\text{head}:: \begin{cases} \text{head} = \emptyset, & \text{head}' \cdot \text{element} = f \land \text{head}' \cdot \text{next}' \sqsubseteq \text{head}_0 \uparrow \land \\ \neg \text{head}' \cdot \text{isNull} \land \text{head} = \emptyset \end{cases}$$

- Object Specification Strengthen Postcondition (7.19)

- Object Specification Weaken Precondition (7.18)

$$\text{head}:: \begin{cases} \text{head}' \cdot \text{element} = f \land \text{head}' \cdot \text{next} = \text{head}_0 \land \\ \neg \text{head}' \cdot \text{isNull} \land \text{head} = \emptyset \end{cases}$$

- Object Specification Strengthen Postcondition (7.19)

$$\text{head}:: \begin{cases} \text{head}' \sqsubseteq (\text{object} \text{ isNull} = \text{false}, \text{element} = f, \text{next} = \text{head}_0 \text{ end}) \land \\ \text{head} = \emptyset \end{cases}$$

- Introduce Reference Clone Field Update (8.9)

$$\text{head} := \text{new} (\text{object} \text{ isNull} = \text{false}, \text{element} = f, \text{next} = \text{head} \text{ end})$$

A similar refinement script can be used to develop the pop method.

### 8.7 Summary

This chapter has introduced new syntax, i.e., aliasing annotations and reference specifications, that are designed to maintain aliasing information and present it in a concise form. It has also introduced two new techniques that ease the development of programs with a reference semantics. The applicability of these techniques is limited if aliasing information is not known. Consequently, the consistent use of aliasing annotations throughout the development of a program is vital.
Chapter 9

Object-Oriented Program Adaptation

This chapter presents an example in which an object-oriented program can be iteratively specified and refined. Section 9.1 introduces a program adaptation technique developed for the classical refinement calculus by Groves [Gro98]. This technique involves the incremental enhancement of a program using a simultaneous execution operator. This operator allows a new, enhanced specification to be given as a combination of the original program and the enhancements. The technique is then extended to an object-oriented paradigm and this extension is illustrated by example in Section 9.2.

9.1 Simultaneous Execution

Mahony [Mah99] presents the definition of a simultaneous execution operator that allows multiple statements to be executed simultaneously. The operator is monotonic and preserves code\(^1\) for statements that modify disjoint sets of variables. Back and von Wright [BvW97, BvW99] provide an alternative presentation of the work involving a similar product operator. Back and Butler [BB95] also note the possibility for application of the product operator in the treatment of inheritance.

Mahony [Mah99] introduces the syntax \(\text{MT}_{v \rightarrow z} v\) for monotonic predicate transformers on program variables \(v\) (in lieu of the \(\text{Ptrans}_{v \rightarrow z} v\) syntax introduced in this thesis). When only a subset \((z)\) of the program variables \((v)\) are modified, Mahony uses the syntax \(\text{MT}_{v \rightarrow z} v\). To allow a monotonic predicate transformer to modify additional variables (variables other than \(z\)) Mahony introduces \(\overline{w}\)-opening. Given a monotonic predicate transformer \(P : \text{MT}_{v \rightarrow z} v\), \(\overline{w}\)-opening allows arbitrary modification of the additional variables in \(\overline{w}\).

\[
P \oplus \overline{w} \cong P; (\overline{w} \setminus v; \text{True})
\]

Hence the type of \(P \oplus \overline{w}\) is \(\text{MT}_{v \cup \overline{w} \setminus v} v\). Mahony uses the syntax \(A(P)\) to denote generalised assumption. To prevent confusion with Back’s [BvW98] use of the nomenclature

\(^1\)Given executable code fragments, the composition will also be executable.
assumption, the term generalised assertion is used here instead. Informally, the generalised assertion of a predicate transformer is the precondition that must be guaranteed to prevent the predicate transformer from aborting. Formally, it is defined as the weakest precondition with respect to the postcondition True.

\[ A(P) \equiv P(\text{True}) \]

Some evaluations of generalised assertions are as follows.

\[ A(a := x) \equiv \text{True} \]

This assumes that \( x \) is well defined.

\[ A(\neg z : p \land q) \equiv p \neg z \land \neg z_0 \quad \neg z \]

The expression \( \neg (P(\neg (\neg z = \neg z))) \neg z \neg z_0 \neg z \) may be read as ‘when starting in a state denoted by \( z_0 \) and executing \( P \), it is not certain that the state \( z \) cannot be reached,’ or alternatively, it is possible that state \( z \) can be reached. Two calculations of generalised effects, for assignments and specifications, are as follows.

\[ E(x := e) \equiv x = (e[x \setminus x_0]) \]

\[ E(\neg z : p \land q) \equiv p(\neg z_0, \neg z) \Rightarrow q \]

However, it is the \( \neg \)-opened version of the generalised effects that are useful. For assignments and specifications, these are:

**Theorem 9.1 (Opened Generalised Effect Basic Properties)**  
Proof on page 197

\[ E((x := e) \neg \neg \neg w_0) \equiv x = e[\neg w, \neg x_0] \]

\[ E(\neg z : p \land q) \equiv p(\neg z_0, \neg w_0) \Rightarrow q \neg w_0 \]

A predicate transformer \( Q \) is conjunctive, denoted by \( Q : MT^\land \), if for arbitrary set \( I \) and predicates \( p_{i \in I} \):

\[ Q(\land_{i \in I} p_i) \equiv \land_{i \in I} Q(p_i) \]

All predicate transformers can be refined to a conjunctive predicate transformer as magic is conjunctive and all predicate transformers refine to magic. The demonic choice of predicate transformers maintains conjunctivity.
Definition 9.2 (Least Conjunctive Refinement) The least conjunctive refinement of a predicate transformer \( P \), denoted by \( \square P \), is the demonic choice of all the conjunctive refinements of \( P \) [Mah99, Definition 3.1], i.e.,
\[
\square P \equiv \square \{ Q : MT^\wedge | P \subseteq Q \}
\]

\( \Diamond \)

Law 9.3 (Least Conjunctive Refinement as a Specification) Using \( A(\_\_\_\_\_\_\_\_) \) and \( E(\_\_\_\_\_\_\_\_) \) the least conjunctive refinement of a statement \( P \) can be determined. For \( P : MT_{\vec{v},\vec{w}} \) [Mah99, Corollary 3.14]:
\[
\square P \equiv \exists: [A(P) \land E(P)]
\]
If \( P \) is conjunctive then it is its own least conjunctive refinement. Consequently, for conjunctive \( P \):
\[
P \equiv \exists: [A(P) \land E(P)]
\]
The demonic choice of statements \( P \) and \( Q \), that is \( P \square Q \), has the effect of (demonically) choosing either \( P \) or \( Q \) and ‘executing’ that choice. To effect the ‘execution’ of both \( P \) and \( Q \), Mahony introduces miraculous conjunction. The miraculous conjunction of \( P : MT_{\vec{v},\vec{w}} \) and \( Q : MT_{\vec{u},\vec{v}} \), written as \( P \astimes Q : MT_{\vec{v},\vec{w}} \), is defined as:
\[
P \astimes Q \equiv \exists: [(A(P) \cap A(Q)) \land (E(P) \cap E(Q))]
\]
Unfortunately, the miraculous conjunction operator does not preserve code: it may introduce infeasible code (a miracle). For instance, the miraculous conjunction of \( a := 1 \) and \( a := 2 \) would establish \( a = 1 \land a = 2 \). If, however, the frames \( (\vec{v} \land \vec{w}) \) are disjoint then each variable can only be modified by one side of the miraculous conjunction and consequently code is preserved. This motivates the introduction of the simultaneous execution operator which is formed by constraining the miraculous conjunction operator to disjoint state frames. That is,
\[
P \upharpoonright_{\vec{v},\vec{w}} Q \equiv P \astimes Q
\]
for \( \vec{v} \land \vec{w} = \emptyset \). For example, the simultaneous execution \( a := 1 \mid b := 2 \) assigns to \( a \) the number 1 and to \( b \) the number 2. The frames are omitted when obvious from the context.

Groves [Gro00, Chapter 7][Gro98] presents a technique which, in some situations can reuse previous program developments when the specification of a program changes. This can be achieved by specifying the program in terms of a miraculous conjunction between the original program and a new program representing the changes. Using a collection of refinement laws, the miraculous conjunction is then distributed through the program and eventually removed. This program evolution often reuses many of the original development steps. An example of a refinement rule that removes miraculous conjunction is
\[\text{ Although not the focus of their work, Back and von Wright [BvW97, BvW99] (especially Theorem 7) have provided some analogous rules.}\]
Miraculous Conjunction Serialisation (9.4). This rule allows miraculous conjunction to be serialised.

**Theorem 9.4 (Miraculous Conjunction Serialisation)** Groves [Gro98, p152] permits the replacement of a miraculous conjunction with a sequential composition provided that the variables modified in statement $S$ are not accessed in statement $T$.

$$u \cap \text{sig}(T) = \emptyset$$

$$S_u \bigtriangleup T \subseteq S; T$$

The syntax $\text{sig}(T)$ denotes the *signature* of, or the variables referenced within, $T$.

Using this rule, the code $a := 1 \mid_b b := 2$ can be refined to the sequential composition: $a := 1; b := 2$. However, the code $a := 1 \mid_b b := a + 2$ cannot be refined to $a := 1; b := a + 2$ due to the presence of, as Groves terms it, *interference*. It can, however, be refined as follows.

$$a := 1 \mid_b b := a + 2$$

$\sqsupseteq$ Commutativity

$$b := a + 2 \mid_a a := 1$$

$\sqsupseteq$ Miraculous Conjunction Serialisation (9.4)

$$b := a + 2; a := 1$$

When extending simultaneous execution for use within the object-oriented refinement calculus, the constraint that the *frames* are disjoint becomes troublesome. That is, the object frames $o.a$ and $o.b$ are not disjoint as they require the modification of the same variable; for a value semantics this is $o$. Consequently a simultaneous execution operator which permits object frames must be defined in a similar manner to miraculous conjunction. For a value object $o$ with disjoint fields $a$ and $b$:

$$P_{o.a \mid o.b} Q \cong o.a, o.b :: \left[ A(P) \land A(Q), E(P \triangleleft o.b) \land E(Q \triangleleft o.a) \right]$$

given $P \triangleleft o.b \cong P; o.b :: [\text{True}]$ and $Q \triangleleft o.a \cong Q; o.a :: [\text{True}]$. Hence $P$ is ‘opened’ so that it can modify $o.b$ and $Q$ is ‘opened’ so that it can modify $o.a$.

Although both sides of the operator modify $o$, they are restricted to modifying disjoint parts of $o$. Thus the two sides do not contradict each other and infeasibility is avoided. Consequently, the preservation of code is maintained by this definition.

Similarly, for a semantics for references, both sides of the simultaneous execution are required to modify the *store* variable (at index $o$). Care is required, however, as aliasing may allow both sub-statements of a simultaneous execution operator to update the same store location despite the disjointness of the object frames. One solution is to ensure that the variables in the object frames are not aliased.
The simultaneous execution operator integrates with the definition of refinement (4.32) to allow code to be refined by simultaneously executing it in conjunction with a chaotic statement.

**Theorem 9.5 (Introduce Chaotic Simultaneous Execution)**  Proof on page 262
Any conjunctive code can be refined by simultaneously executing it with \( \bar{w} : [\text{True}] \) for variables \( \bar{w} \) of types \( \bar{W} \). For conjunctive \( pt \) defined on environment \( \bar{p} \), where \( \bar{p} \) and \( \bar{w} \) are disjoint:

\[
\text{pt} \sqsubseteq \text{pt} | \bar{p} \bar{w} : [\text{True}]
\]

In the classical (non-object-oriented) refinement calculus, to use the refinement \( \text{pt} | \bar{p} \bar{w} : [\text{True}] \) in place of \( \text{pt} \) requires reducing its environment to \( \bar{p} \). Reducing the environment to \( \bar{p} \) can be achieved by encapsulating the program fragment within a variable block:

\[
[\text{var } \bar{w} : \bar{W} \bullet \text{pt} | \bar{p} \bar{w} : [\text{True}]]
\]

**Theorem 9.6 (Introduce Object Field for Legacy)**  Proof on page 262
When a new field is introduced into an object, the Introduce Chaotic Simultaneous Execution (9.5) rule can be used to enhance the pre-existing methods with the ability to modify the new fields.

Given an object with fields \( f_{1,i} \) and conjunctive methods \( m_{1,k} \), adding new fields \( f_{i+1,j+j} \) (for \( j \geq 0 \)) permits the original methods to be simultaneously executed with chaos on those fields. For new method labels \( m_{k+1,k+l} \) (for \( l \geq 0 \)), field values \( f_{v1,i+j} \) of types \( F_{1,i+j} \), and methods \( mv_{1,k+l} \):

\[
\begin{align*}
\text{object} & \quad \text{field } f_{h \in 1,i} : F_h := f_{vh} \\
& \quad \text{method } m_{h \in 1,k} = mv_h \\
\text{end} \\
\sqsubseteq & \\
\text{object} & \quad \text{field } f_{h \in 1,i+j} : F_h := f_{vh} \\
& \quad \text{method } m_{h \in 1,k} = mv_h f_{1,i} [f_{i+1,i+j} f_{i+1,i+j} \ddots : [\text{True}]} \\
& \quad \text{method } m_{h \in k+1,k+l} = mv_h \\
\text{end}
\end{align*}
\]

### 9.2 Simultaneous Execution Example

The following is based on an example presented by Mikhajlova and Sekerinski [MS97] in which a banking system is enhanced with new functionality that allows all transactions of

\[\text{Morgan [Mor94] uses the nomenclature } \text{choose} \text{ for a demonic choice statement. Back and von Wright [BvW98] use } \text{choose} \text{ to indicate an angelic choice statement. To avoid confusion, the specification } \bar{w} : [\text{True}] \text{ is used in this thesis as a statement that demonically chooses variables } \bar{w}.\]
an account to be logged. Here the program adaptation techniques of Groves [Gro98] are
modified to an object-oriented paradigm and used to present an alternative approach.

The main class of the banking system is the Account class. The Account class has
fields for the account owner’s name and the balance of the account. It also has a con-
structor and methods for depositing money, withdrawing money, returning the name of
the owner and for returning the balance of the account.

```haskell
class Account is
  field owner : Name
  field balance : Currency
  method Account(value name : Name, value amount : Currency) ≜
    owner := name | balance := amount
  method Deposit(value amount : Currency) ≜
    {amount > 0}; balance := balance + amount
  method Withdraw(value amount : Currency) ≜
    {amount > 0 ∧ amount ≤ balance};
    balance := balance − amount
  method Owner(result name : Name) ≜
    name := owner
  method Balance(result cur : Currency) ≜
    cur := balance
end
```

The original example extended the functionality of this class with a transaction log. This
included an extended interface so that the transaction log can record additional details
about the transactions, such as the date it occurred. Mikhajlova and Sekerinski extended
the interface by providing the class with a wrapper class which contained an extended
interface. An alternative is to provide rules for adding parameters to methods. The addi-
tion of result parameters is intuitively a refinement as they can be simply ignored by the
callee of the method. Value parameters can be added to a method’s interface provided that
they are associated with a default value. When a method is called without specifying the
value parameter, the default value would be used[^4]. Using this approach the Deposit and
Withdraw methods could be extended with a value parameter, when : Date. The default
value for the when parameter is omitted from the class yet is assumed to be the given
constant no-date : Date.

Individual transactions are recorded within Transaction objects.

```haskell
class Transaction is
  field amount : Currency
  field date : Date
  method Transaction(value a : Currency, value d : Date) =
    amount, date := a, d
end
```

[^4]: To efficiently use this technique, a parameter naming mechanism would provide additional benefits
over the positional parameter style employed here. Such a mechanism would allow more flexible control
over which parameters are provided.
A field (transactions) containing a sequence of Transaction objects is added to the Account class. This allows the methods to be refined to simultaneous executions that update transactions. In this manner the program is developed incrementally though an iterative process of specification and refinement.

\[\exists\] Introduce Object Field for Legacy (9.6)
Object Specification Strengthen Postcondition (7.19)

```
class Account2 is
  field owner : Name
  field balance : Currency
  field transactions : seq(Transaction)

  method Account(value name : Name, value amount : Currency) =
    ( owner := name | balance := amount ) | transactions := ()

  method Deposit(value amount : Currency, value when : Date) =
    ( {amount > 0}; balance := balance + amount ) |
    transactions :: [\exists t : Transaction •
      transactions = transactions0 \cup \{t\} 
      t.amount = amount \land t.date = when ]

  method Withdraw(value amount : Currency, value when : Date) =
    ( {amount > 0 \land amount \leq balance} ) |
    balance := balance - amount |
    transactions :: [\exists t : Transaction •
      transactions = transactions0 \cup \{t\} 
      t.amount = -amount \land t.date = when ]

  method Owner(result name : Name) = name := owner
  method Balance(result cur : Currency) = cur := balance
end
```

Using Update Object Method (7.5) in conjunction with the rules from Chapter 7, this
class is refined to

\[\text{\texttt{\#5: Refine Classes (7.50)}}\]
\textbf{class} Account\textsubscript{3} \textbf{is}
\begin{itemize}
  \item [field] owner : Name
  \item [field] balance : Currency
  \item [field] transactions : seq(Transaction)
\end{itemize}
\textbf{method} Account\texttt{(value name : Name, value amount : Currency)} =
\begin{align*}
  \{ & \text{owner := name} \mid \text{balance := amount} \} \mid \\
  \text{transactions := \langle \rangle}
\end{align*}
\textbf{method} Deposit\texttt{(value amount : Currency, value when : Date)} =
\begin{align*}
  \{ & \{\text{amount > 0}\}; \text{balance := balance + amount} \} \mid \\
  \text{\textbf{var} t : Transaction} \bullet \\
  \text{t.call Transaction(amount, when);} \\
  \text{transactions := transactions \triangleleft \langle t \rangle}
\end{align*}
\textbf{method} Withdraw\texttt{(value amount : Currency, value when : Date)} =
\begin{align*}
  \{ & \{\text{amount > 0} \land \text{amount \leq balance}\} \} \mid \\
  \text{\textbf{var} t : Transaction} \bullet \\
  \text{t.call Transaction(\text{-}amount, when);} \\
  \text{transactions := transactions \triangleleft \langle t \rangle}
\end{align*}
\textbf{method} Owner\texttt{(result name : Name)} = \text{name := owner}
\textbf{method} Balance\texttt{(result cur : Currency)} = \text{cur := balance}
\textbf{end}

If \textit{super} is given a semantics such that it refers to the immediate superclass\textsuperscript{5} of a subclass, then the ‘original’ code within this could be refined to calls on \textit{super}. Finally, the simultaneous execution operators can be reduced to sequential compositions using Theorem 9.4. This step is not required, however, as the simultaneous executions can be considered as code. Using Class Introduction (7.49) and Subclasses (7.51), the following subclass could

\textsuperscript{5}Assuming multiple inheritance is not supported.
be introduced.

```plaintext
subclass Account′ of Account is
  field transactions : seq(Transaction)
  method Account(value name : Name, value amount : Currency) =
    super.call Account(name, amount);
    transactions := ⟨⟩
  method Deposit(value amount : Currency, value when : Date) =
    super.call Deposit(amount, from);
    var t : Transaction ⋆
    t.call Transaction(amount, when);
    transactions := transactions ^⟨t⟩
  ␥]
  method Withdraw(value amount : Currency, value when : Date) =
    super.call Withdraw(amount, when);
    var t : Transaction ⋆
    t.call Transaction(−amount, when);
    transactions := transactions ^⟨t⟩
  ␥]
end
```

Back and Butler [BB95] briefly discuss providing inheritance with a semantics such that the methods of a subclass are, by definition, simultaneously executed with the methods of the original (super)class. This example could use such an approach as the enhancements do not require alteration of the original class. Such an approach would elicit the following subclass where the super-calls have been replaced by the syntax +.

```plaintext
subclass Account′ of Account is
  field transactions : seq(Transaction)
  method Account(value name : Name, value amount : Currency) =
    +transactions := ⟨⟩
  method Deposit(value amount : Currency, value when : Date) =
    +[var t : Transaction ⋆
      t.call Transaction(amount, when);
      transactions := transactions ^⟨t⟩
    ␥]
  method Withdraw(value amount : Currency, to : Name, when : Date) =
    +[var t : Transaction ⋆
      t.call Transaction(−amount, when);
      transactions := transactions ^⟨t⟩
    ␥]
end
```

The discussion of the use of super-calls on page 102 is reiterated here: one could argue that the discontinuity that super-calls introduce outweigh their usefulness as code reuse mechanisms. They are used as the ‘goto’ of object-orientation, i.e., they are effective both at code reuse and reducing understanding.
9.3 Summary

This chapter has exemplified the use of the simultaneous execution operator in the context of an object-oriented refinement calculus. The simultaneous execution operator has been shown to be well suited to incrementally improving classes, allowing a new, enhanced specification to be given as a combination of the original program and the enhancements. This is aligned with the objectives of subclassing.
Chapter 10

Conclusions

This thesis has covered the important issues regarding the integration of a refinement calculus with the object-oriented programming paradigm. The aim of the thesis has been the development of a scalable, calculational-style development method. Specifically, this is to be achieved through the provision of an object-oriented refinement calculus.

10.1 Summary

After an introduction to the background and literature (Chapters 1, 2 and 3), the thesis provides a basis for an object-oriented refinement calculus. This basis involves the definition of a typed refinement calculus (Chapter 4). Using the typed refinement calculus, an abstraction for objects that include predicate transformers is developed (Chapter 5). Constructs to define and use predicate transformer objects are defined and to address practical concerns, a semantics for references is provided (Chapter 6). Additional constructs are defined for use with the semantics for references. Since the semantics for references is a definitional extension of the semantics for values, the resulting language could be termed extra-wide-spectrum as it allows the specification and development of both value and reference semantics programs within a single semantic framework.

Chapter 7 builds on the basis to develop an object-based refinement calculus. This is achieved through the provision of an object-refinement relation, object-data-refinement relation and refinement rules to allow the reification of specifications. A class-based refinement calculus is then provided as a definitional extension.

In Chapter 8 the reference semantics is augmented with notations and techniques to help address the extra complexity of developing programs with a semantics for references. Finally, Chapter 9 involves the extension of Groves’ [Gro98] program adaptation technique for use within an object-oriented refinement calculus.
10.2 Results and Impacts

One of the main issues for object-oriented refinement calculi is object representation. The research summarised in Chapter 3 has typically provided object representations in which the methods are separated from the fields. This approach has been taken to simplify the associated object types. However, it (at least in theory) breaks the encapsulation of objects and requires an additional dynamic dispatch mechanism to be added to the semantics. The object model presented in Chapter 5 provides an example of the alternative approach in which methods are embedded in objects. Both approaches still deserve attention as it is not yet clear which is better. For instance, the ‘method embedded’ model has a more succinct approach to modelling dynamic dispatch, yet has difficulties modelling multiple inheritance.

For object representations there appears to be, in general, a tradeoff between complexity and completeness. Simpler models do not provide support for the addition of new attributes to subclasses, e.g., [MS97] and the models that do are more complex, e.g., [Nau94b]. Additionally, none of the existing models provide support for all desired object-oriented features. For instance, the subtyping of arbitrary binary methods cannot, in general, be handled. Research on object-type theory indicates that this may be an enduring problem. Judicious use of the typecase construct, as seen in Chapter 5, shows that in certain circumstances, the problem can be solved.

Chapter 5 provides an object representation based on a particular object calculus, namely Abadi and Cardelli’s $\text{FOb}_{\leq \mu}$ [AC96]. One theoretical constraint arising from this choice is that unbounded non-determinism is not supported. The thesis has been developed, however, to allow the current object model to be easily replaced by another that does not have this constraint. Further work is required to develop other object calculi that do not have this constraint.

A heterogeneously typed framework was developed in Chapter 4. This framework included an innovative open world view of refinement. This definition permits refinements such as $a, b := 1, 1 \subseteq a, b, c := 1, 1, 1$ where the environment of the first statement involves only $a$ and $b$. Opening the refinement relation in this manner is intuitive. The assignment $a, b := 1, 1$ in an environment containing only $a$ and $b$ can be considered to be an assignment that also arbitrarily updates $c$. Since the environment does not constrain $c$ to a type, it may be assigned, for example, an integer or even a boolean value. By allowing variables outside the environment to be modified, this definition of refinement allows programs to be more easily combined. This is a step towards a scalable refinement calculus. The ‘open’ definition is shown (Chapter 9) to be conducive to the incremental extension of classes through subclassing. Additionally, the ‘open world’ view allows classes to be replaced by subclasses using algorithmic refinement. Several of the object-oriented refinement calculi summarised in Chapter 3 have required the use of data refinement to permit the substitution of objects. This is aesthetically displeasing as the subclass in-
stances subsume the type of the superclass and hence no change of type is required. A data-refinement-based substitution is required, however, to allow the client to use the new attributes of the subclass.

Several of the object-oriented refinement calculi summarised have focussed on the provision of a class-based language. This thesis indicates that a better approach is to develop an object-based calculus and definitionally extend it to a class-based calculus. This provides greater flexibility, allowing both object-based and class-based programming techniques.

The characterising property of the object-refinement relation(s) presented in this thesis is the ability to substitute an object with an object-refinement. To achieve this monotonicity within statements, a solution inspired by Naumann [Nau94a] is to weaken predicates to those monotonic in object-refinement. Given an object variable \( o \) and constant object \( e \), an example of a non-monotonic predicate is \( o = e \). Given a state in which this predicate holds, that is, when \( o \) is equal to \( e \), it is not possible to replace the value of \( o \) with an object-refinement and still maintain the validity of the predicate. Consequently the use of non-monotonic predicates is not conducive to practical development. Instead, a predicate such as \( e \sqsubseteq^c o \) should be used. In this case, replacing \( o \) with an object-refinement maintains the validity of the predicate.

It was also necessary to prove that the statements do not introduce non-monotonic predicates. This required some restrictions on the predicates used within specification statements. The proofs were originally performed by King [Kin99]. This thesis contributes to this area by proving that the constraints imposed on the specification’s predicates can be relaxed. Previously, the postcondition was required to be anti-monotonic for the variables. This prevented specification statements such as

\[
o: [p \sqsubseteq^c o]
\]

for object variables \( o \) and \( p \) as the postcondition is not anti-monotonic in \( o \). We have relaxed this constraint so that the postcondition is required to be anti-monotonic for variables not in the frame.

Fortunately the restriction to monotonic predicates has limited practical impact on the development of programs. In fact, all results of the classical refinement calculus are maintained. A similar restriction is also required in other object-oriented refinement calculi. For example, as summarised on page 21 of this thesis, Utting and Robinson deal with this problem in the context of their calculus.

Another contribution made by the thesis is the unique modelling of private attributes. Rather than use an ancillary mechanism to hide attributes as is done within several of the object-oriented refinement calculi presented in Chapter 3, the core object-oriented data abstraction mechanism, subsumption, is reused. Using the \textit{private} syntax constructs an object instance of the true (\textit{private}) object type but stores it in a variable that is of the \textit{public} type. Since the client is not permitted access to the \textit{private} type, subsumption
successfully hides the private attributes.

For some of the object-oriented refinement calculi discussed in this thesis, object fields are introduced so that each field is modelled directly by a variable in the state. Consequently, the fields can be used directly in specification statements. For calculi (including the one presented by this thesis) that encapsulate all of an object’s fields into a single entity, modifying one field requires the modification of the variable representing the entire object. Additional constraints are needed to prevent the unwanted modification of fields (see Example 3.1). The use of the object specifications introduced in this thesis have proven to be beneficial in making these constraints implicit, thereby improving the clarity of the specifications and allowing fine-grained control of the modification of attributes.

To address practical concerns, a semantics for references was introduced. As references are supported through definitional extension, the semantics for references in this thesis co-exists and complements the semantics for values. An innovative technique is introduced in Chapter 8 that provides a link between the two semantics. Semantic conversion allows the development of a program using the simpler semantics for values. This program can then be converted to a program with a semantics for references. Additionally, a code segment in a reference semantics program can be temporarily converted to a value semantics for easier development. The conversions between the semantics are straightforward. Proof of the soundness of the technique is provided.

Another innovative technique is presented that removes the superfluous aliasing in a reference semantics program, allowing the developer to concentrate on the program’s ‘true’ aliasing. Coalesced programming allows the temporary mapping of aliases onto a single unique or primary variable. After aliasing has been removed, the semantic conversion technique can be used to allow the program (segment) to be developed using the simpler semantics for values. Like semantic conversion, the conversions to and from coalesced programs is straightforward. Proof of the soundness of coalesced programming is provided.

In summary, this thesis has addressed the important issues of an object-oriented refinement calculus and has progressed towards the development of a scalable, calculational development method.

10.3 Future Work

The work of the thesis could be extended by an attempt to identify alternative object models that satisfy the subtyping properties identified in Chapter 5 yet also supports unbounded nondeterminism. This thesis has examined the use of a functional object calculi ($\text{FOb}_{\mu}$). It would be worthwhile investigating the use of imperative object calculi [AC96].

An object-oriented refinement calculus with a semantics for references is more practical (than one with a semantics for values) due to the popularity of object-oriented lan-
guages with reference-based semantics. A calculus with an exclusive semantics for references may also be theoretically more elegant. For instance, for the semantics for values the object-refinement relation (and the object type) is required to be recursive due to the nesting of objects. In a semantics for references, the nesting of objects is flattened out into a (one-level) store. Consequently, object-refinement (and the object type) no longer need to be recursive. This observation could possible be used to substantially simplify the object representation and hence the resulting calculus. However, it would most likely be more difficult to develop programs in such a calculus.

The collection of refinement rules that this thesis presents is by no means comprehensive. There are many classical refinement rules that could be lifted. Additionally, this thesis’s treatment of aliasing annotations, reference specifications and object specifications are proofs of concepts. It is anticipated that further investigation of their properties would be fruitful.

Mikhajlova [Mik99] illustrates the formal use of the Wrapper design pattern [GHJV95]. Investigation of the formalisation of design patterns may lead to a form of ‘design refactoring’ where, for example, the designer applies high level transformation rules to modify a class hierarchy to introduce flexibility. It is envisaged that most structural design patterns would be amenable for such formalisation, though it is unclear whether other design pattern styles, e.g., behavioural, would be too. Hopefully such investigations would bear fruit to true object-oriented refinement rules, rather than the mere object flavoured refinement rules that currently exist.

Another interesting topic to investigate would be the viability of associating object-refinements with a graphical notation (such as UML [JBR99, PJ99]). It would be desirable to provide a formal link between a graphical notation and the object-oriented refinement calculus, thereby producing graphical refinement rules. For example, one obvious graphical refinement rule would be the introduction of a subclass as shown in Figure 10.1.

Although a multiple store approach was introduced, little attention has been paid to addressing the movement of aliases between the stores and the resulting impacts on development. Further consolidation of the multiple store approach is required.
Bibliography


[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, 1995. {6,158}


[Pre95] W. Pree. *Design Patterns for Object-Oriented Software Development*. Addison Wesley, 1995. {6}


{20,28,43,105,179,182,193}


Appendix A

Object Calculus

This appendix presents the complete object calculus used for the construction of the object-oriented refinement calculus. Section A.1 presents the axiomisation of the calculus. Section A.2 extends the calculus to support variance annotations. Section A.3 presents an axiomisation of records.

A.1 Axiomisation

The calculus is $\text{FOb}_{\llangle \mu}$ as presented by Abadi and Cardelli [AC96]. They also provide a denotational semantics for this calculus which guarantees its soundness [AC96, Chapter 14]. The calculus $\text{FOb}_{\llangle \mu}$ was chosen for its features: the support of variance annotations, recursive and function types. A functional calculus was preferred over an imperative calculus as it more easily integrates with the refinement calculus. The syntax is sugared for readability within the context of the thesis.

Abadi and Cardelli incrementally construct the calculus. This incremental construction is mirrored in this appendix. The root of the $\text{FOb}_{\llangle \mu}$ calculus is the $F_{\llangle}$ calculus. The $F_{\llangle}$ calculus comprises theory fragments concerning:

- environments ($\Delta_\llangle$),
- functions ($\Delta_\rightarrow$),
- subtyping ($\Delta_{\llangle}$),
- environment subtyping ($\Delta_{\llangle X}$),
- function subtyping ($\Delta_{\llangle \rightarrow}$), and
- universal types ($\Delta_{\llangle V}$).

The theory fragment for universal types ($\Delta_{\llangle V}$) is omitted as they are not used within the thesis.

\[
F_{\llangle} = \Delta_\llangle \cup \Delta_\rightarrow \cup \Delta_{\llangle} \cup \Delta_{\llangle X} \cup \Delta_{\llangle \rightarrow} \cup \Delta_{\llangle V}
\]

\[
\Delta_\llangle
\]
**Axiom A.1 (Env ☐)** The empty environment is well formed.

\[ \emptyset \vdash \emptyset \]

**Axiom A.2 (Env x)** Given a well formed type \( A \) and fresh \( x \), then typing \( x \) as \( A \) maintains a well formed environment.

\[ E \vdash A \quad x \not\in \text{dom}(E) \]

\[ E \cup \{ x : A \} \vdash \emptyset \]

**Axiom A.3 (Val x)** \( x : A \) is a truth if it is in the environment.

\[ E \cup \{ x : A \} \vdash \emptyset \]

\[ E \cup \{ x : A \} \vdash x : A \]

**Axiom A.4 (Type Arrow)** Function types are well formed.

\[ A \quad B \]

\[ A \rightarrow B \]

**Axiom A.5 (Val Fun)** The function \( \lambda(x : A) \) is of type \( A \rightarrow B \) if \( b : B \) given \( a : A \).

\[ E \cup \{ x : A \} \vdash b : B \]

\[ E \vdash \lambda(x : A)b : A \rightarrow B \]

**Axiom A.6 (Val Appl)** The type of a function application.

\[ f : A \rightarrow B \quad a : A \]

\[ f(a) : B \]

**Axiom A.7 (Sub Refl)** Subtyping is reflexive.

\[ A \]

\[ A \preceq A \]

**Axiom A.8 (Sub Trans)** Subtyping is transitive.

\[ A \preceq B \quad B \preceq C \]

\[ A \preceq C \]

**Axiom 4.3 (Val Subsumption)** The general rule of subsumption:

\[ a : A \quad A \preceq B \]

\[ a : B \]

**Axiom A.9 (Type Top)** \( \top \) is a well formed type.

\[ \top \]

\[ \top \]

\[ \text{Top} \]
Axiom A.10 (Sub Top) All types are subtypes of Top.

\[ \Delta \ni A \quad A \preceq \text{Top} \]

Axiom A.11 (Env X \preceq) Subtyping information can be included in a well-formed environment. For fresh type label X:

\[ E \vdash A \quad X \not\in \text{dom}(E) \]

\[ E \cup \{ X \preceq A \} \vdash \emptyset \]

Axiom A.12 (Type X \preceq) Subtypes are well-formed types.

\[ E \cup \{ X \preceq A \} \vdash \emptyset \]

Axiom A.13 (Sub X) Subtypes in the environment are truth.

\[ E \cup \{ X \preceq A \} \vdash X \preceq A \]

Axiom 4.10 (Sub Arrow) Function types subtype covariantly in the result and contravariantly in the argument.

\[ A' \preceq A \quad B \preceq B' \]

\[ A \rightarrow B \preceq A' \rightarrow B' \]

The $F_{\preceq}$ calculus is extended with theory fragments for objects ($\Delta_{ob}$) and their subtypings ($\Delta_{\preceq ob}$) to form the $FOb_{\preceq}$ calculus.

\[ FOb_{\preceq} = F_{\preceq} \cup \Delta_{ob} \cup \Delta_{\preceq ob} \]

Axiom 4.1 (Type Object) The object type $Obj \{ \ i, i, \ 1 \rightarrow n \} \rightarrow \beta_i \}$ is well-formed if the type of each of its attributes is well-formed.

\[ \text{for all } (i \in 1..n) \bullet \beta_i \]

\[ Obj \{ \ i, i, \ 1 \rightarrow n \} \rightarrow \beta_i \}

Axiom 4.2 (Val Object) The type of an object is determined by the types of its attributes. Given $\gamma \preceq Obj F$:

\[ \text{for all } (l \in \text{dom}(F)) \bullet E \cup \{ x : \gamma \} \vdash \text{body}_l : F(l) \]

\[ E \vdash \text{object} \{ \ l, l \in \text{dom}(F) \bullet l = \varsigma(x : \gamma) \text{body}_l \} : \gamma \]

Axiom 4.7 (Val Select) The type of a method invocation (field selection) is the type of the method body associated with the label being selected.

\[ c : Obj F \quad l \in \text{dom} F \]

\[ c \circ l : F(l) \]
**Axiom A.14 (Val Update)**  Object updates are typed as follows.

\[
E \vdash c : \text{Obj } F \quad E \cup \{x : \text{Obj } F\} \vdash b : F(l) \quad l \in \text{dom } F
\]

\[
E \vdash c \bowtie I \iff \zeta(x : \text{Obj } F)b : \text{Obj } F
\]

\[
\Delta_{\neg \exists \mu}\]

**Axiom 4.4 (Sub Object)**  Subtyping of object types occurs when the subtype has more attributes.

\[
F \subseteq G \quad \text{Obj } G
\]

\[
\text{Obj } G \preceq \text{Obj } F
\]

The \(\text{FOb}_{\preceq \mu}\) calculus is extended with recursive types (\(\Delta_{\preceq \mu}\)) to form the \(\text{FOb}_{\preceq \mu}\) calculus.

\[
\text{FOb}_{\preceq \mu} \equiv \text{FOb}_{\preceq \mu} \cup \Delta_{\preceq \mu}
\]

**Axiom A.15 (Type Rec \(\preceq\))**  Recursive types are well formed.

\[
E \cup \{X \preceq \text{Top}\} \vdash A
\]

\[
E \vdash \mu(X)A
\]

**Axiom A.16 (Sub Rec)**  Recursive types subtype covariantly if their bounds subtype covariantly.

\[
E \vdash \mu(X)A \quad E \vdash \mu(Y)B \quad E \cup \{Y \preceq \text{Top}\} \cup \{X \preceq Y\} \vdash A \preceq B
\]

\[
E \vdash \mu(X)A \preceq \mu(Y)B
\]

**Axiom A.17 (Val Fold)**  The following provides a type for recursive foldings. Given

\[
A \equiv \mu(X)B\{X\}:
\]

\[
b : B[X\setminus A]
\]

\[
\text{fold}(A, b) : A
\]

**Axiom A.18 (Val Unfold)**  The following provides a type for recursive unfoldings. Given

\[
A \equiv \mu(X)B\{X\}:
\]

\[
a : A
\]

\[
\text{unfold}(a) : B[X\setminus A]
\]

The \(\text{FOb}_{\preceq \mu}\) calculus is extended with existential types (\(\Delta_{\preceq \exists}\)) to form the \(\text{FOb}_{\preceq \mu}\) calculus. Existential types are partially abstracted types. A review of existentially quantified types is provided by Abadi and Cardelli [AC96, p173]. Existential types are not used directly by the thesis. They are used only in Section A.2 for the interpretation of variantly annotated object types.

\[
\text{FOb}_{\preceq \mu} \equiv \text{FOb}_{\preceq \mu} \cup \Delta_{\preceq \exists}
\]

\[
\Delta_{\preceq \exists}\]
APPENDIX A. OBJECT CALCULUS

Axiom A.19 (Type Exists) Existential types are well formed.
\[ E \cup \{ X \preceq A \} \vdash B \]
\[ E \vdash \exists (X \preceq A)B \]

Axiom A.20 (Sub Exists) Existential types subtype covariantly in their argument and result types.
\[ E \vdash A \preceq A' \quad E \cup \{ X \preceq A \} \vdash B \preceq B' \]
\[ E \vdash \exists (X \preceq A)B \preceq \exists (X \preceq A')B' \]

Axiom A.21 (Val Pack) The following types existential packings.
\[ C \preceq A \]
\[ b[X \backslash C] : B[X \backslash C] \]
\[ \langle \text{pack } X \preceq A = C \text{ with } b\{X\} : B\{X\} \rangle : \exists (X \preceq A)B\{X\} \]

Axiom A.22 (Val Open) The following types existential openings.
\[ E \vdash c : \exists (X \preceq A)B \quad E \vdash D \quad E \cup \{ X \preceq A \} \cup \{ x : B \} \vdash d : D \]
\[ E \vdash \langle \text{open } c \text{ as } X \preceq A, x : B \text{ in } d \vdash D \rangle : D \]

Supplementing these axioms are various equational theory fragments. The equational theories for
- objects (\(\Delta_{=\text{Ob}}\) and \(\Delta_{\equiv\text{Ob}}\)),
- equational symmetry and transitivity (\(\Delta_{=}\)),
- equational reflexivity (\(\Delta_{=x}\)),
- functions (\(\Delta_{=\_x}\)),
- subsumption (\(\Delta_{=\_}\)), and
- recursion (\(\Delta_{=\mu}\)) are presented.

\(\Delta_{=\text{Ob}}\)

Axiom 4.5 (Eq Object) Two object instances are equivalent if all respective attributes are equivalent.
\[ \text{for all } (i \in 1..n) \bullet E \cup \{ x : \gamma \} \vdash b_i =_{\beta_i} b'_i \]
\[ E \vdash \text{object } \{ i \in 1..n \bullet l_i = \zeta(x : \gamma) b_i \} =_{\gamma} \text{object } \{ i \in 1..n \bullet l_i = \zeta(x : \gamma) b'_i \} \]
where \(\gamma \equiv \text{Obj } \{ i \in 1..n \bullet l_i : \beta_i \} \). The premise ensures the equivalence of the attributes given that the self parameter (x) has type \(\gamma\).

Axiom 4.8 (Eq Select) Given two equivalent objects, selecting the same label on each produces two objects that are also equivalent.
\[ c =_{\text{Obj } F} c' \quad l \in \text{dom } F \]
\[ c \oplus l =_{F(l)} c' \oplus l \]
Axiom A.23 (Eq Update) Given $\gamma \cong \text{Obj } F$
\[ E \vdash a =_\gamma a' \quad E \cup \{ x : \gamma \} \vdash b =_{F(l)} b' \quad l \in \text{dom } F \]
\[ E \vdash a \circ l =_\gamma \varsigma(x : \gamma) b =_\gamma a' \circ l =_\gamma \varsigma(x : \gamma) b' \]

Axiom 4.6 (Eval Select) Given $\gamma \cong \text{Obj } \{ i \in 1..n \bullet l_i : \beta_i \}$ and
\[ c \cong \text{object } \{ i \in 1..n \bullet l_i = \varsigma(x : \gamma) b_i \} \]
then
\[ c : \gamma \quad j \in 1..n \]
\[ c \circ l_j =_{\beta_j} b_j[x'/c] \]

Axiom 4.9 (Eval Update) Given $\gamma \cong \text{Obj } F$ and $c \cong \text{object } f$:
\[ E \vdash c : \gamma \quad E \cup \{ x : \gamma \} \vdash b : F(l) \quad l \in \text{dom } F \]
\[ E \vdash (c \circ l =_\gamma \varsigma(x : \gamma) b) =_\gamma \text{object } (f \oplus \{ l = \varsigma(x : \gamma) b \}) \]

Axiom A.24 (Eq Sub Object) Given $\gamma \equiv \text{Obj } \{ i \in 1..n \bullet l_i : \beta_i \}$,
\[ \zeta \equiv \text{Obj } \{ i \in 1..n + m \bullet l_i : \beta_i \}, \text{for } m \geq 0, \]
for all $(i \in 1..n) \bullet E \cup \{ x : \gamma \} \vdash b_i : \beta_i$
for all $(j \in n + 1..n + m) \bullet E \cup \{ x : \zeta \} \vdash b_j : \beta_j$
\[ E \vdash \text{object } \{ i \in 1..n \bullet l_i = \varsigma(x : \gamma) b_i \} =_\gamma \text{object } \{ i \in 1..n + m \bullet l_i = \varsigma(x : \zeta) b_i \} \]

Axiom A.25 (Eq Symm) Equality on a type is symmetric.
\[ a =_\alpha b \]
\[ b =_\alpha a \]

Axiom A.26 (Eq Trans) Equality on a type is transitive.
\[ a =_\alpha b \quad b =_\alpha c \]
\[ a =_\alpha c \]

Axiom A.27 (Eq x) Equivalence is reflexive.
\[ E \cup \{ x : A \} \vdash \circ \]
\[ E \cup \{ x : A \} \vdash x =_A x \]

Axiom A.28 (Eq Fun) Two functions are equivalent if their bodies are equivalent.
\[ E \cup \{ x : A \} \vdash b =_B b' \]
\[ E \vdash \lambda(x : A) b =_{A \rightarrow B} \lambda(x : A) b' \]
Axiom A.29 (Eq Appl) Given two equivalent functions and two equivalent arguments, the application of the functions to the arguments is equivalent.

\[
\frac{b =_{A \rightarrow B} b' \quad a =_{A} a'}{b(a) =_{B} b'(a')}
\]

Axiom A.30 (Eval Beta) The following property is beta-conversion for functions.

\[
\frac{E \vdash \lambda(x : A) b : A \rightarrow B \quad E \vdash a : A}{E \vdash \lambda(x : A) b(a) =_{B} b[x \backslash a]}
\]

Axiom A.31 (Eval Eta) The following property is eta-conversion for functions.

\[
\frac{E \vdash b : A \rightarrow B \quad x \notin \text{dom}(E)}{E \vdash \lambda(x : A) b(x) =_{A \rightarrow B} b}
\]

Axiom A.32 (Eq Subsumption) Two objects equal on a type are also equal on a supertype.

\[
\frac{a =_{\alpha} b \quad \alpha \preceq \beta}{a =_{\beta} b}
\]

Axiom A.33 (Eq Top) All objects are equal on type \(\text{Top}\).

\[
\frac{a : \alpha \quad b : \beta}{a =_{\text{Top}} b}
\]

Axiom A.34 (Eq Fold) The foldings of two equivalent objects are also equal. For \(A \cong \mu(X)B\{X\},\)

\[
\frac{b =_{B[X \backslash A]} b'}{\text{fold}(A, b) =_{A} \text{fold}(A, b')}
\]

Axiom A.35 (Eq Unfold) The unfoldings of two equivalent recursive objects are also equal. For \(A \cong \mu(X)B\{X\},\)

\[
\frac{a =_{A} a'}{\text{unfold}(a) =_{B[X]} \text{unfold}(a')}
\]

Axiom A.36 (Eval Fold) Unfolding and refolding a recursive object produces an object that is equal to the original. For \(A \cong \mu(X)B\{X\},\)

\[
\frac{a : A}{\text{fold}(A, \text{unfold}(a)) =_{A} a}
\]

Axiom A.37 (Eval Unfold) Folding and unfolding an object produces another object that is equal to the original. For \(A \cong \mu(X)B\{X\},\)

\[
\frac{b : B[X \backslash A]}{\text{unfold}(\text{fold}(A, b)) =_{B[X \backslash A]} b}
\]
A.2 Split Method Interpretation

The $FOb_{\leq \mu}$ calculus can be extended with variance annotations as described by Abadi and Cardelli [AC96, p269] and represented here. This extension is an interpretation of the calculus $FOb_{\leq \mu}$ in terms of another that does not support objects. This interpretation generates the axioms provided earlier in this appendix. The inclusion of variance annotations is in addition to the interpretation. The entire interpretation is provided despite our sole interest in the provision of variance annotations.

Object types are interpreted as recursive existential types with a special $r$ component that is bound recursively to the entire record. The methods are split into those that may be updated ($upd$) and those that may be invoked or selected ($sel$). This interpretation is analogous to the recoup technique [AC96, Section 15.6].

For distinct $l_i$

$$Obj \{ l_i : B_i \} \equiv \mu(Y) \exists(X \preceq Y)C\{X\}$$

where

$$C\{X\} \equiv \langle r : X, l^{sel}_i : X \to B_i^{j \in 1..n}, l^{upd}_i : (X \to B_i) \to X^{i \in 1..n} \rangle$$

object \{ $l_i = \varsigma(x_i : A)b_i$ \} \equiv

let rec create($y_j : A \to B_i^{j \in 1..n}$) : $A =$
fold($A, pack X = A$ with

$$\langle r = create(y_j^{j \in 1..n}, l^{sel}_i = y_j^{j \in 1..n},$$
$$l^{upd}_i = \lambda(y'_j : A \to B_i)create(y_j^{j \in 1..i-1}, y'_j, y_k^{k \in (i+1..n)}_{i \in 1..n})$$
$$: C\{X\} \rangle \in create(\lambda(x_j : A)b_i^{j \in 1..n})$$

Method invocation opens and unfolds the recursive existential type and applies the recursive $r$ component to the method to be invoked.

For $j \in 1..n$

$$o_{A \odot l_j} \equiv open unfold(o) as X \preceq A, p : C\{X\} in p_{i \odot l^{sel}_j}(p_{i \odot r}) : B_j$$

Method update opens and unfolds the recursive existential type and uses function application to construct a new object.

For $j \in 1..n$

$$o_{A \odot l_j} = \varsigma(y : A)b \equiv open unfold(o) as X \preceq A, p : C\{X\} in p_{l^{upd}_j}(\lambda(y : A)b) : A$$

As described by Abadi and Cardelli [AC96, p270], variance annotations can be mapped onto these object types. Invariant attributes $l^o$ are translated to two record components $l^{sel}$ and $l^{upd}$. Covariant attributes $l^+$ are translated to the record components $l^{sel}$ and contravariant attributes $l^-$ are translated to the record components $l^{upd}$. Using this interpretation the following variance annotation axioms (rules of the interpretation) are validated. These axioms are discussed further by Abadi and Cardelli [AC96, p110].
Axiom A.38 (Type Object Variance) Given variance annotations $\nu_i \in 1..n$, the object type $\text{Obj} \{ i \in 1..n \cdot l_i^n \rightarrow \beta_i \}$ is well-formed if the type of each of its attributes is well-formed.

$$\text{for all } (i \in 1..n) \cdot \beta_i$$
$$\text{Obj} \{ i \in 1..n \cdot l_i^n \rightarrow \beta_i \}$$

Axiom A.39 (Sub Object Variance) Variantly annotated objects subtype objects with more attributes or according to their components’ variance annotations.

$$\text{for all } (i \in 1..n) \cdot \nu_i B_i \simeq \nu_i' B'_i \text{ for all } i \in n + 1..n + m \cdot B_i$$
$$\text{Obj} \{ i \in 1..n + m \cdot l_i \nu_i : B_i \} \simeq \text{Obj} \{ i \in 1..n \cdot l_i \nu_i' : B'_i \}$$

Axiom A.40 (Sub Invariant) Invariantly annotated entities (e.g., objects, functions) subtype reflexively only.

$$B$$
$$cB \iff c'B$$

Axiom A.41 (Sub Covariant) Covariantly annotated entities subtype covariantly.

$$B \simeq B' \quad \nu \in \{\circ, +\}$$
$$\nu B \simeq +B'$$

Axiom A.42 (Sub Contravariant) Contravariantly annotated entities subtype contravariantly.

$$B' \simeq B \quad \nu \in \{\circ, -\}$$
$$\nu B \simeq -B'$$

Axiom A.43 (Val Object Variance) The type of an object is determined by the types of its methods. Given $\gamma \equiv \text{Obj} \{ i \in 1..n \cdot l_i^n \rightarrow \beta_i \}$

$$\text{for all } (i \in 1..n) \cdot E \cup \{x : \gamma \} \vdash \text{body}_i : \beta_i$$
$$E \vdash \text{object} \{ i \in 1..n \cdot l_i = \varsigma(x : \gamma) \text{body}_i \} : \gamma$$

Axiom A.44 (Val Select Variance) To prevent unsoundness selection is restricted to invariant and covariant annotated methods.

$$c : \text{Obj} \{ i \in 1..n \cdot l_i^n \rightarrow \beta_i \} \quad \nu_j \in \{\circ, +\} \quad j \in 1..n$$
$$c \circ l_j : \beta_j$$

Axiom A.45 (Val Update Variance) To prevent unsoundness update is restricted to invariant and contravariant annotated methods. Given $\gamma \equiv \text{Obj} \{ i \in 1..n \cdot l_i^n \rightarrow \beta_i \}$:

$$E \vdash c : \gamma \quad E \cup \{x : \gamma\} \vdash b : \beta_j \quad \nu_j \in \{\circ, -\} \quad j \in 1..n$$
$$E \vdash c \circ l_j \equiv \varsigma(x : \gamma)b : \gamma$$
A.3 Record Calculus

The following axioms and definitions are from a theory of records ($\Delta_{Rcd}$) provided by Abadi and Cardelli [AC96, Section 8.6.1, p106].

Axiom A.46 (Type Record) The record type $\{i \in 1..n \mid l_i : B_i\}_{\text{RT}}$ is well-formed provided each component type is well-formed. For distinct $l_i \in 1..n$:

$$\frac{\text{for all } (i \in 1..n) \cdot B_i}{\{i \in 1..n \mid l_i : B_i\}_{\text{RT}}}$$

Axiom 4.11 (Sub Record) Given an environment where $\text{rfun}_{\text{RT}}$ and $\text{rfun'}_{\text{RT}}$ are well-formed record types,

$$\frac{E \vdash \text{dom}(\text{rfun'}) \subseteq \text{dom}(\text{rfun})}{\text{for all } (i \in \text{dom}(\text{rfun'})) \cdot E \vdash \text{rfun}(i) \equiv \text{rfun'}(i)}$$

$$\frac{\text{for all } (i \in \text{dom}(\text{rfun}) \setminus \text{dom}(\text{rfun'})) \cdot E \vdash \text{rfun}(i)}{E \vdash \text{rfun}_{\text{RT}} \equiv \text{rfun'}_{\text{RT}}}$$

Axiom A.47 (Val Record) The record $\{i \in 1..n \mid l_i = b_i\}_{\text{RT}}$ has the type $\{i \in 1..n \mid l_i : B_i\}_{\text{RT}}$ provided each component of the record has the appropriate type.

$$\frac{\text{for all } (i \in 1..n) \cdot b_j : B_i}{\{i \in 1..n \mid l_i = b_i\}_{\text{RT}} \vdash \{i \in 1..n \mid l_i : B_i\}_{\text{RT}}}$$

Axiom A.48 (Val Record Select) The type of a record selection is identified by the label within the record type of the record being selected.

$$a : \{i \in 1..n \mid l_i : B_i\}_{\text{RT}} \quad j \in 1..n$$

$$\frac{a.l_j : B_j}{\text{a} \cdot \{i \in 1..n \mid l_i = b_i\}_{\text{RT}} \vdash a \cdot \{l_j = b_j\}_{\text{RT}}}$$

For readability, we define $\oplus$ for both records and record types:

$$\{i \in 1..n \mid l_i = b_i\}_{\text{RT}} \oplus \{l_j = b_j\}_{\text{RT}} \equiv \{i \in (1..n) \setminus \{j\} \mid l_i = b_i, l_j = b_j\}_{\text{RT}}$$

$$\{i \in 1..n \mid l_i : B_i\}_{\text{RT}} \oplus \{l_j : B_j\}_{\text{RT}} \equiv \{i \in (1..n) \setminus \{j\} \mid l_i : B_i, l_j : B_j\}_{\text{RT}}$$

Definition A.49 (Functional Record Update) Functional record update is defined in terms of record construction and record selection.

$$a.l_j := b \equiv a \oplus \{l_j = b\}_{\text{RT}}$$

for $a$ of type $\{i \in 1..n \mid l_i : B_i\}_{\text{RT}}$. 
Axiom A.50 (Val Record Update) The type of a record update is the type of the record being updated augmented (or overwritten) with the type of the component that is being updated.

\[ a : \{ i \in 1..n \cdot l_i : B_i \}_{xt} \quad b : B \quad j \in 1..n \]

\[ a.l_j := b : \{ i \in 1..n \cdot l_i : B_i \}_{xt} \oplus \{ l_j : B \}_{xt} \]

Axiom A.51 (Eq Record Update) Given two equivalent records, and two equivalent objects, updating one record with the first object is equivalent to updating the second record with the second object. Let \( \gamma \equiv \{ i \in 1..n \cdot l_i : B_i \}_{xt} \) and \( \gamma' \equiv \gamma \oplus \{ l_j : B \}_{xt} \):

\[ a =_{\gamma} a' \quad b =_{B} b' \quad j \in 1..n \]

\[ a.l_j := b =_{\gamma} a'.l_j := b' \]

Axiom A.52 (Eval Record Update) Updating a record is equivalent to constructing a new one by copying most components of the original record and replacing the one to be updated. Given \( a \equiv \{ i \in 1..n \cdot l_i = b_i \}_x \) and \( \gamma \equiv \{ i \in 1..n \cdot l_i : B_i \}_{xt} \) and \( \gamma' \equiv \gamma \oplus \{ l_j : B \}_{xt} \):

\[ a : \gamma \quad b : B \quad j \in 1..n \]

\[ a.l_j := b =_{\gamma} a \oplus \{ l_j = b \}_x \]

Axiom A.53 (Eq Record) Two records are equivalent if their components are equivalent. Given \( \gamma \equiv \{ i \in 1..n \cdot l_i = b_i \}_x \):

\[
\text{for all } (i \in 1..n) \cdot b_i =_{B_i} b'_i \\
\{ i \in 1..n \cdot l_i = b_i \}_x =_{\gamma} \{ i \in 1..n \cdot l_i = b_i' \}_x
\]

Axiom A.54 (Eq Sub Record) Given \( \gamma \equiv \{ i \in 1..n \cdot l_i = b_i \}_x \):

\[
\text{for all } (i \in 1..n + m) \cdot b_i : B_i \\
\{ i \in 1..n + m \cdot l_i = b_i \}_x =_{\gamma} \{ i \in 1..n + m \cdot l_i = b_i \}_x
\]

Axiom A.55 (Eq Record Select) The results of two record selections are equivalent if the records being selected are equivalent as well as the label being selected. Given \( \gamma \equiv \{ i \in 1..n \cdot l_i : B_i \}_{xt} \):

\[ a =_{\gamma} a' \quad j \in 1..n \]

\[ a.l_j =_{B_j} a'.l_j \]

Axiom A.56 (Eval Record Select) A record selection is equivalent to the appropriate component from the record. Given \( a \equiv \{ i \in 1..n \cdot l_i = b_i \}_x \):

\[ a : \{ i \in 1..n \cdot l_i : B_i \}_{xt} \quad j \in 1..n \]

\[ a.l_j =_{B_j} b_j \]
Appendix B

A Formal System of Objects

This appendix provides a reference of the additional laws, definitions and theorems used in the thesis. Section B.1 presents miscellaneous predicate calculus definitions and properties. Section B.2 provides miscellaneous predicate calculus laws for use in proofs. Section B.3 presents the semantics of statements (and associated refinement rules) not covered in Chapter 4. A number of the refinement rules have been taken from Morgan’s work [Mor94]. Section B.4 provides the classical data refinement rules used in this thesis. Finally, Section B.5 presents additional object-refinement properties not listed in Chapter 7.

B.1 Predicate Calculus

This section provides the typed semantics and properties for predicate conjunction and disjunction. This material was not covered in Chapter 4 but is required for proofs.

**Definition B.1 (Predicate Conjunction)** Given state types $\alpha$ and $\beta$, predicate conjunction is defined as the lifting of boolean conjunction on the greatest lower bound of these types.

\[
p_1 : \text{Pred} \quad p_2 : \text{Pred} \quad (\alpha \sqcap \beta) \\
p_1 \land p_2 \triangleq \lambda s : (\alpha \sqcap \beta) \cdot p_1(s) \land p_2(s)
\]

The following theorem presents the typing of predicate conjunction.

**Theorem B.2 (Val Conjunction)** Proof on page 177

\[
p_1 : \text{Pred} \quad p_2 : \text{Pred} \quad (\alpha \sqcap \beta) \\
(p_1 \land p_2) : \text{Pred} (\alpha \sqcap \beta)
\]
Proof of B.2 from p176

\[(p_1 \land p_2) : \text{Pred} (\alpha \sqcap \beta)\]
\[\iff\] Predicate Conjunction (B.1), provided \((\alpha \sqcap \beta)\) is well-formed.
\[(\lambda s : (\alpha \sqcap \beta) \bullet p_1(s) \land p_2(s)) : \text{Pred} (\alpha \sqcap \beta)\]
\[\iff\] Val Fun (A.5)
\[s : (\alpha \sqcap \beta) \vdash (p_1(s) \land p_2(s)) : \mathbb{B}\]
\[\iff\] Val Appl (A.6)
\[p_1 : \text{Pred} (\alpha \sqcap \beta) \land p_2 : \text{Pred} (\alpha \sqcap \beta)\]
\[\iff\] Sub Predicate (4.18), Val Subsumption (4.3)
\[p_1 : \text{Pred} \alpha \land p_2 : \text{Pred} \beta\]

\[\Box\]

Definition B.3 (Predicate Disjunction) Given state types \(\alpha\) and \(\beta\), predicate disjunction is defined as the lifting of boolean disjunction on the greatest lower bound of these types.

\[p_1 : \text{Pred} \alpha \quad p_2 : \text{Pred} \beta \quad (\alpha \sqcap \beta)\]

\[p_1 \lor p_2 \equiv \lambda s : (\alpha \sqcap \beta) \bullet p_1(s) \lor p_2(s)\]

\[\Box\]

The following theorem presents the typing of predicate disjunction.

Theorem B.4 (Val Disjunction)

\[p_1 : \text{Pred} \alpha \quad p_2 : \text{Pred} \beta \quad (\alpha \sqcap \beta)\]

\[\langle p_1 \lor p_2 \rangle : \text{Pred} (\alpha \sqcap \beta)\]

Proof

The proof is similar to that of Val Conjunction (B.2).

\[\Box\]

B.2 Predicate Calculus Laws

The laws in this section are basic predicate calculus laws derived from [Mor94, p258, Appendix A] and/or [Mar67].

Law B.5 (Double Implication)

\[A \Rightarrow (B \Rightarrow C) \equiv (A \land B) \Rightarrow C \equiv B \Rightarrow (A \Rightarrow C)\]
Law B.6 (Existential Quantification One Point Rule) Provided \( x \) is not free in \( E \),
\[
A[x\setminus E] \equiv (\exists x \bullet x = E \land A)
\]

Law B.7 (Universal Quantification One Point Rule) Provided \( x \) is not free in \( E \),
\[
A[x\setminus E] \equiv (\forall x \bullet x = E \Rightarrow A)
\]

Law B.8 (Universal de Morgan)
\[
\neg (\forall x \bullet A) \equiv (\exists x \bullet \neg A)
\]

Law B.9 (Extended Commutativity of Disjunction)
\[
(\exists x \bullet (\exists y \bullet A)) \equiv (\exists x, y \bullet A)
\]

Law B.10 (Universal Quantification Weak Distribution)
\[
(\forall x \bullet A \Rightarrow B) \Rightarrow (\forall x \bullet A) \Rightarrow (\forall x \bullet B)
\]
The name ‘weak’ denotes the use of \( \Rightarrow \) rather than \( \equiv \).

Law B.11 (Existential Quantification Weak Distribution)
\[
(\exists x \bullet A \land B) \Rightarrow (\exists x \bullet A) \land (\exists x \bullet B)
\]

Law B.12 (Partially Superfluous Universal Quantification Implication)
\[
(\forall x \bullet N \Rightarrow B) \equiv N \Rightarrow (\forall x \bullet B)
\]
for \( x \) not free in \( N \).

Law B.13 (Partially Superfluous Universal Quantification)
\[
(\forall x \bullet N \land B) \equiv N \land (\forall x \bullet B)
\]
for \( x \) not free in \( N \).

Law B.14 (Partially Superfluous Existential Quantification)
\[
(\exists x \bullet N \land B) \equiv N \land (\exists x \bullet B)
\]
for \( x \) not free in \( N \).

Law B.15 (Universal Quantification Elimination) For any term \( E \)
\[
(\forall x \bullet A) \Rightarrow A[x\setminus E]
\]
If \( A \) is true for all \( x \), then it is true for \( x \) being equal to \( E \).

Law B.16 (Existential Quantification Introduction)
\[
A[x\setminus E] \Rightarrow (\exists x \bullet A)
\]
If \( A \) is true for \( x \) being \( E \), then it is true for at least one \( x \).
B.3 Statements

This section provides the typed semantics for the statements used in this thesis that are not covered in Chapter 4. The properties and refinement rules used in this thesis are also reproven for the definitions provided.

The definitions in this section are based in part on the those provided by Sekerinski [Sek96], Morgan [Mor94], and Back and von Wright [BvW98].

Skip

Definition B.17 (Skip) \(\text{skip}\) is the identity predicate transformer. Like \(\text{abort}\), \(\text{skip}\) is actually a family of predicate transformers.

\[
\text{skip}_\alpha \equiv \lambda p : \text{Pred } \alpha \cdot p
\]

The subscript is omitted when obvious from the context.

\[\diamond\]

Theorem B.18 (Val Skip) Proof on page 199 Given state type \(\alpha\) the \(\text{skip}_\alpha\) predicate transformer types as

\[
\text{skip}_\alpha : \text{Ptrans } \alpha \alpha
\]

Assertion

Definition B.19 (Assertion) The assertion \(\{p\}_\beta\) specifies a predicate that can be assumed to hold. The programmer is not responsible for establishing an assertion. Given a predicate \(p : \text{Pred } \alpha\), the assertion \(\{p\}_\beta\) behaves like \(\text{skip}_\beta\) when \(p\) is true yet like \(\text{abort}_\beta\) when \(p\) is false. Morgan [Mor94, p254] uses the terminology assumption for assertions. Unfortunately, Back [BvW98, p192] uses assumption for what is termed, in this thesis, a guard, or coercion. To avoid confusion, the term assumption has no meaning with respect to statements in this thesis. Assertions are defined on a state type \(\beta\) when the greatest lower bound of \(\beta\) and \(\alpha\) exists.

\[
\frac{p : \text{Pred } \alpha \quad (\alpha \sqcap \beta)}{\{p\}_\beta \equiv \lambda q : \text{Pred } \beta \cdot p \land q}
\]

\[\diamond\]

The subscript is omitted when obvious from the context.

Theorem B.20 (Val Assertion) Proof on page 199 The assertion \(\{p\}_\beta\) is typed as follows:

\[
\frac{p : \text{Pred } \alpha \quad (\alpha \sqcap \beta)}{\{p\}_\beta : \text{Ptrans } \beta (\alpha \sqcap \beta)}
\]
The state type $\beta$ is chosen as required by the context. If $\beta$ is chosen to be $\alpha$ then the assertion types as

\[
\{p\}_\alpha : \text{Ptrans } \alpha \ \alpha
\]

and the well-formedness of $(\alpha \sqcap \alpha)$ reduces to the well-formedness of $\alpha$.

**Theorem B.21 (Weaken Assertion)**  Proof on page 203  Given predicates $p : \bot \text{ Pred } P$ and $q : \text{Pred } Q$, and state type $\alpha$ such that $(\text{Pred } P \sqcap (\text{Pred } Q \sqcap \alpha))$,

\[
p \Rightarrow q \\
\{p\}_\alpha \sqsubseteq \{q\}_\alpha
\]

Alternatively, an assertion can be removed entirely.

**Law B.22 (Remove Assertion)** According to Morgan [Mor94, p308], any assertion can be refined by skip. For predicate $p : \bot \text{ Pred } P$, and state type $\alpha$ such that $(\alpha \sqcap \text{Pred } P)$,

\[
\{p\}_\alpha \sqsubseteq \text{skip}_\alpha
\]

**Law B.23 (Absorb Assertion)** An assertion before a specification can be absorbed into its precondition.

\[
\{\text{pre}'\}_\alpha ; \ w : \{\text{pre} , \ \text{post}\}_\alpha = w : \{\text{pre}' \land \text{pre} , \ \text{post}\}_\alpha
\]

This law is drawn from [Mor94, p298,1.8].

**Guard**

**Definition B.24 (Guard)** The guard $[p]_\beta$, also known as coercion, specifies a predicate $p$ that the programmer is responsible for establishing. After the execution of the guard, the predicate $p$ is true. If $p$ were true beforehand, then the guard skips, if $p$ was not true, then the guard behaves as magic. If $p$ is a predicate on state type $\alpha$, then the guard $[p]_\beta$ is defined when the greatest lower bound of $\alpha$ and $\beta$ exists.

\[
p : \text{Pred } \alpha \ (\alpha \sqcap \beta) \\
[p]_\beta \triangleq \lambda q : \text{Pred } \beta \bullet p \Rightarrow q
\]

\[\diamond\]

**Theorem B.25 (Val Guard)**  Proof on page 200  The guard $[p]_\beta$ is typed as follows:

\[
p : \text{Pred } \alpha \ (\alpha \sqcap \beta) \\
[p]_\beta : \text{Ptrans } \beta \ (\alpha \sqcap \beta)
\]

The state type $\beta$ is chosen as required by the context in an analogous manner to that for assertions.
Law B.26 (Absorb Guard) A guard following a specification can be absorbed into its postcondition. For state type $\alpha$,

$$w : [\text{pre} \ , \ \text{post}]_\alpha ; \ [\text{post'}]_\alpha = w : [\text{pre} \ , \ \text{post} \land \text{post'}]_\alpha$$

This law is drawn from [Mor94, p298,17.2].

Law B.27 (Introduce Assertion) For state type $\alpha$,

$$[\text{post}]_\alpha \sqsubseteq [\text{post}]_\alpha ; \ \{\text{post}\}_\alpha$$

This law is drawn from [Mor94, p305,17.19].

Update

Definition B.28 (Update) Given a state transformer $st : \alpha \rightarrow \beta$ the update operator $\langle st \rangle$ lifts it to a predicate transformer.

$$\langle st \rangle_\beta \triangleq \lambda p : \text{Pred} \ \beta \bullet (\lambda s : \alpha \bullet p(st \ s))$$

Theorem B.29 (Val Update Statement) Proof on page 200 Update types as a predicate transformer from the range type of the state transformer function to its domain type.

$$st : \alpha \rightarrow \beta \quad \langle st \rangle_\beta : \text{Ptrans} \ \beta \alpha$$

The predicate transformers constructed using state transformers are deterministic. Consequently, this is a useful technique for modelling assignments.

Assignment

Definition B.30 (Assignment) Given state type $\alpha$ including distinct variables $i_j : I_{j \in 1..m}$, and expressions $e_j : I_{j \in 1..m}$ well defined in $\alpha$, assignment can be defined syntactically using Morgan’s definition [Mor94, p250].

$$\langle i_1, \ldots, i_m := e_1, \ldots, e_m \rangle_\alpha \triangleq (\lambda p : \text{Pred} \ \alpha \bullet p[i_1, \ldots, i_m \setminus e_1, \ldots, e_m])$$

Theorem B.31 (Val Assignment) Proof on page 201 Given state type $\alpha$ including distinct variables $i_j : I_{j \in 1..m}$, and expressions $e_j : I_{j \in 1..m}$ well defined in $\alpha$, an assignment statement types as follows:

$$\langle i_1, \ldots, i_m := e_1, \ldots, e_m \rangle_\alpha : \text{Ptrans} \ \alpha \alpha$$

Law B.32 (Introduce Assignment) Assume state type $\alpha$ including $w$ and $x$ and expression $E$ well defined in $\alpha$. If $\text{pre} \Rightarrow \text{post}[w \setminus E]$, then [Mor94]

$$w, x : [\text{pre} \ , \ \text{post}]_\alpha \sqsubseteq (w := E)_\alpha$$
Enter

**Definition B.33 (Enter)** The enter predicate transformer is a deterministic (functional) state update that introduces a new component and uses the precondition state to calculate initial values. It can be thought of intuitively as the opening clause of a local variable block. The enter statement can be provided a typed definition using Sekerinski’s [Sek96] definition as a basis. For state type \( \alpha \) where the variable being introduced \( v : V \) is not in \( \text{dom}(\alpha) \).

\[
\begin{align*}
v \not\in \text{dom}(\alpha) \quad \text{initv} : V \\
(\text{enter} \ v : V := \text{initv})_\alpha \triangleq \langle \lambda s : \alpha \bullet s \cup \{v = (\text{eval initv} \ s)\} s \rangle (\alpha \setminus \{v\}_{\text{st}})
\end{align*}
\]

\[\checkmark\]

**Theorem B.34 (Val Enter)** _Proof on page 201_ For state types ‘compatible’ with state type \( \{v : V\}_{\text{st}} \), the enter predicate transformer types as follows:

\[
\begin{align*}
v \not\in \text{dom}(\alpha) \quad \text{initv} : V \\
(\text{enter} \ v : V := \text{initv})_\alpha : \text{Ptrans} (\alpha \setminus \{v : V\}_{\text{st}}) \alpha
\end{align*}
\]

It is known that \( (\alpha \setminus \{v : V\}_{\text{st}}) \) is well-formed from \( v \not\in \text{dom}(\alpha) \).

Exit

**Definition B.35 (Exit)** The exit statement is complementary to the enter statement as it intuitively encapsulates the closing bracket of a local variable block. It is a deterministic state update that removes a particular state variable.

\[
\begin{align*}
v \not\in \text{dom}(\alpha) \\
(\text{exit} \ v : V)_\alpha \triangleq \langle \lambda s : (\alpha \setminus \{v : V\}_{\text{st}}) \bullet (\sqcap t : \alpha | t =_\alpha s \bullet t) \rangle_\alpha
\end{align*}
\]

\[\checkmark\]

**Theorem B.36 (Val Exit)** _Proof on page 201_ Given state type \( \alpha \),

\[
(\text{exit} \ v : V)_\alpha \triangleq : \text{Ptrans} \alpha (\alpha \setminus \{v : V\}_{\text{st}})
\]

Demonic Choice

**Definition B.37 (Generalised demonic choice)** Demonic choice can be generalised to an arbitrary number of alternatives. For an index type \( I \) the demonic choice is defined as follows.

\[
\begin{align*}
\text{for all} \ (i \in I) \bullet \ pt_i : \text{Ptrans} \alpha_i \beta_i (\sqcap \{j \in I \bullet \beta_j\}) \\
(\sqcap i \in I \bullet pt_i) \triangleq \lambda p \in \text{Pred} (\sqcup \{j \in I \bullet \alpha_j\}) \bullet (\forall i \in I \bullet pt_i \ p)
\end{align*}
\]
Sequential Composition

**Definition B.38 (Sequential Composition)** Sequential composition is functional composition of predicate transformers. Sequential composition for two predicate transformers is defined when the postcondition state type of the first is a subtype of the precondition state type of the second:

\[
\begin{align*}
pt_1 : Ptrans & \quad \alpha \rightarrow \delta \quad pt_2 : Ptrans & \quad \beta \quad \gamma \quad \alpha \subsetneq \gamma \\
\langle pt_1; pt_2 \rangle & \equiv \lambda p : \text{Pred} \quad \beta \quad \bullet \quad pt_1 (pt_2 p)
\end{align*}
\]

\[\Diamond\]

**Theorem B.39 (Val Sequential Composition)** Proof on page 201 Sequential composition, when defined, types as a predicate transformer from the postcondition state type of 'second' predicate transformer to the precondition state type of the 'first'.

\[
\begin{align*}
pt_1 : Ptrans & \quad \alpha \rightarrow \delta \quad pt_2 : Ptrans & \quad \beta \quad \gamma \quad \alpha \subseteq \gamma \\
\langle pt_1; pt_2 \rangle & : Ptrans \quad \beta \rightarrow \delta
\end{align*}
\]

**Law B.40 (Introduce Sequential Composition)** For fresh constants \(X\) and state type \(\alpha\),

\[
\begin{align*}
w, x: [pre \quad , \quad post]_{\alpha} & \\
\equiv & \\
\left[\left[ \begin{align*}
& \con X \bullet \\
& x: [pre \quad , \quad mid]_{\alpha}; \\
& w, x: [mid[x_0 \setminus X] \quad , \quad post[x_0 \setminus X]]_{\alpha}
\end{align*} \right] \right]
\end{align*}
\]

and formula \(mid\) that does not contain initial variables other than \(x_0\).

This rule is presented by Morgan [Mor94, Law 8.4, p310].

**Local Variable Block** A typed definition for the local variable block can be provided using the previously defined enter and exit statements.

**Definition B.41 (Local Variable Block)** Given predicate transformer \(P : Ptrans (\alpha \sqcap \{a : A\})_{sr} \quad (\alpha \sqcap \{a : A\})_{sr}\), and \(av : A\),

\[
\left[\left[ \var a : A = av \bullet P \right] \right]_{\alpha} \equiv (\text{enter} \quad a : A := av)_{\alpha}; \quad (\text{exit} \quad a : A)_{\alpha}
\]

The initialisation is deterministic as it is based on the deterministic enter statement, which in turn, is based on a deterministic state update. \(P\), however, may be nondeterministic.

An alternative definition for a non-deterministically initialised version is provided by Morgan [Mor94].

\[
\left[\left[ \var a : A \bullet P \right] \right]_{\alpha} \equiv \lambda post : \text{Pred} \quad \alpha \bullet (\forall a : A \bullet P \quad post)
\]

provided \(a\) is not free in \(post\).

The subscript is omitted when obvious from the context.
Theorem B.42 (Val Local Variable Block)  Proof on page 202  Given predicate transformer $P : Ptrans (\alpha \cap \{a : A\}{_{ST}}) (\alpha \cap \{a : A\}{_{ST}})$,  

$[\text{var } a : A = av \bullet P]_{\alpha} : Ptrans \alpha \alpha$

Logical Constants  Logical constants are angelically chosen values. They are used to help refine specifications, yet they are not code and must be removed for the code to be executable.

Definition B.43 (Logical Constant) Morgan [Mor94] defines logical constants as follows. For state type $\alpha$, and $pt : Ptrans \alpha \alpha$,  

$[\text{con } lcon \bullet pt]_{\alpha} \cong \lambda post : Pred \alpha \bullet (\exists lcon \bullet pt post)$

provided $lcon$ is not free in $post$.

The subscript may be omitted when obvious from the context.

Theorem B.44 (Val Logical Constant)  Proof on page 202  For state type $\alpha$, and $pt : Ptrans \alpha \alpha$,  

$[\text{con } lcon \bullet pt]_{\alpha} : Ptrans \alpha \alpha$

provided $lcon$ is not free in $post$.

Law B.45 (Remove Logical Constant)  A logical constant may be removed provided it is not used in a program. For state type $\alpha$, and $pt : Ptrans \alpha \alpha$,  

$[\text{con } lcon \bullet pt]_{\alpha} \subseteq pt$

provided $lcon$ does not occur in $pt$.

Law B.46 (Fix Initial Value) For state type $\alpha$, any term $E$, well defined in $\alpha$, such that $pre \Rightarrow E \in T$, and fresh name $c$, [Mor94]  

$w : [pre \wedge c = E \bullet post]_{\alpha}$

$\subseteq$

$[[\text{con } c : T \bullet$

$w : [pre \wedge c = E \bullet post]_{\alpha}$

$]]$
APPENDIX B. A FORMAL SYSTEM OF OBJECTS

Specification

Definition B.47 (Specification Statement) The specification statement $v : [pre, post]_\alpha$ alters variables $v$ so that $post$ holds, given that $pre$ held before the specification statement is executed. If it is not possible for $post$ to be established by modifying $v$ (e.g., $post$ were False) then the specification statement is equivalent to magic. Morgan [Mor94] defines specifications as follows. For state type $\alpha$, $pre : Pred \alpha$, and $post : Pred \alpha$,

$$v : [pre, post]_\alpha \equiv \lambda p : Pred \alpha \bullet pre \land (\forall v \bullet post \Rightarrow p)(v_0 \setminus v)$$

The predicate (relation) $post$ may contain references to the initial values of $v$ using the subscript notation: $v_0$.

The subscript is omitted when obvious from the context.

Theorem B.48 (Val Specification) Proof on page 202 For state type $\alpha$, $v \in \text{dom}(\alpha)$, $pre : Pred \alpha$, and $post : Pred \alpha$,

$$v : [pre, post]_\alpha : \text{Ptrans} \alpha \alpha$$

Specifications that use zero-subscripted variables can be reduced to a specification encapsulated by a logical constant [Mor94, Abbreviation 8.2, p69].

Law B.49 (Initial Variable) For fresh $X$, and state type $\alpha$,

$$w : [pre, post]_\alpha \equiv \{ [\text{con} X \bullet w : [pre \land x = X, post[x_0 \setminus X]]_\alpha] \}$$

Law B.50 (Expand Frame) The frame can be expanded using an expand frame rule [Mor94, Law 8.3, p69]. For state type $\alpha$ where $x \in \text{dom}(\alpha)$,

$$w : [pre, post]_\alpha \equiv w, x : [pre, post \land x = x_0]_\alpha$$

Law B.51 (Contract Frame) The frame can be contracted using a contract frame law [Mor94, Law 5.4]. For state type $\alpha$,

$$w, x : [pre, post]_\alpha \sqsubseteq w : [pre, post[x_0 \setminus x]]_\alpha$$

Law B.52 (Simple Specification) An assignment can easily be translated to a specification statement. For state type $\alpha$,

$$i : [i = (e)[i \setminus i_0]]_\alpha \equiv (i := e)_\alpha$$

provided $e$ contains no $i$ [Mor94].

More generally, for any relation $\odot$,

$$i : \odot e \equiv i : [i \odot (e)[i \setminus i_0]]$$
Law B.53 (Weaken Precondition) The precondition of a specification can be weakened. For state type $\alpha$,

$$ \text{pre} \Rightarrow \text{pre}' $$

$$ v : \text{pre} , \text{post} \alpha \subseteq v : \text{pre}' , \text{post} \alpha $$

This law is drawn from [Mor94, p314.1.12].

Law B.54 (Strengthen Postcondition) The postcondition of a specification can be strengthened. For state type $\alpha$,

$$ \text{post} \Rightarrow \text{post}' $$

$$ v : \text{post}' \alpha \subseteq v : \text{post} \alpha $$

The precondition can also be used to strengthen the postcondition [Mor94, Law 5.1].

$$ \text{pre}[v \setminus v_0] \land \text{post} \Rightarrow \text{post}' $$

$$ v : \text{pre} , \text{post}' \alpha \subseteq v : \text{pre} , \text{post} \alpha $$

Law B.55 (Introduce Local Variable Block) A variable not in the environment can be introduced. For state type $\alpha$, if $x$ does not occur in $w$, $\text{pre}$, $\text{post}$, or $\text{dom} (\alpha)$, then [Mor94]

$$ w : \text{pre} , \text{post} \alpha \subseteq \{ \text{var} x : T \bullet x , w : \text{pre} , \text{post} \}(\alpha \sqcap \{x: \alpha \} ) $$

Alternation

Definition B.56 (Alternation Statement) Given guards $G_{i\in 1..n} : \text{Pred} \alpha$ and predicate transformers $P_{i\in 1..n} : \text{Ptrans} \alpha \alpha$ [Mor94, p251],

$$ \text{if } \{ G_{i\in 1..n} \rightarrow P_i \} \equiv \lambda p : \text{Pred} \alpha \bullet (\bigvee_{i\in 1..n} G_i) \land (\bigwedge_{i\in 1..n} G_i \Rightarrow P_i (p)) $$

\triangleright

Theorem B.57 (Val Alternation) Proof on page 203 Given guards $G_{i\in 1..n} : \text{Pred} \alpha$ and predicate transformers $P_{i\in 1..n} : \text{Ptrans} \alpha \alpha$:

$$ \text{if } \{ G_{i\in 1..n} \rightarrow P_i \} : \text{Ptrans} \alpha \alpha $$

Law B.58 (Alternation) Given state type $\alpha$, guards $G_{i\in 1..n}$, well defined in $\alpha$, and $\text{pre} \Rightarrow GG$ where $GG$ is the disjunction of the guards:

$$ w : \text{pre} , \text{post} \alpha \subseteq \text{if } \{ G_{i\in 1..n} \rightarrow w : \text{pre} \land G_i , \text{post} \}(\alpha) $$
Iteration

**Definition B.59 (Iteration Statement)** Let \( C(p) \) be a program fragment in which the name \( p \) appears. Then

\[
\text{re } p \bullet C(p) \text{ er}
\]

is the least-refined program fix such that \( C(\text{fix}) = \text{fix} \) [Mor94, p255].

Iteration is a special case of the definition of recursion given above. Given guards \( G_{i \in 1..n} : \text{Pred } \alpha \), disjunction of the guards \( GG \), and \( \text{prog}_{i \in 1..n} : \text{Ptrans } \alpha \alpha \),

\[
\text{do } ([G_{i \in 1..n} \rightarrow \text{prog}_i]) \text{ od } \equiv \text{re } p \bullet \text{if } [G_{i \in 1..n} \rightarrow \text{prog}_i; p] \nrightarrow GG \rightarrow \text{skip}_\alpha \text{ fi er}
\]

**Theorem B.60 (Val Iteration)** Proof on page 203

Given guards \( G_{i \in 1..n} : \text{Pred } \alpha \), and \( \text{prog}_{i \in 1..n} : \text{Ptrans } \alpha \alpha \):

\[
\text{do } ([G_{i \in 1..n} \rightarrow \text{prog}_i]) \text{ od } : \text{Ptrans } \alpha \alpha
\]

**Law B.61 (Iteration)** Given the invariant \( \text{inv} \), the integer variant \( V \) and the disjunction of the guards \( GG \) [Mor94]:

\[
w : [\text{inv}, \text{inv} \land \nrightarrow GG] \sqsubseteq \text{do } ([G_{i \in 1..n} \rightarrow w : [\text{inv} \land G_i, \text{inv} \land (0 \leq V < V_0)]] \text{ od}
\]

Neither \( \text{inv} \) or \( GG \) may contain initial variables.

**Parameterised Method Call**

**Theorem B.62 (Result Parameterised Method Call)** Proof on page 225

This rule permits the use of a result parameter in the method being introduced. Assume a formal result parameter \( f \), an actual result parameter \( a \) (both disjoint from \( w \)) where \( a \) is neither \( \text{self} \) or \( o \), and a method \( m \) of object \( o \). For a semantics for values, if

\[
w, f : [\text{pre}, \text{post}[a \setminus f]] \sqsubseteq (o \circ m)[\text{self} \setminus o]
\]

with \( \text{pre} \) containing no \( f \) and neither \( f \) nor \( f_0 \) occurring in \( \text{post} \), then

\[
w, a : [\text{pre}, \text{post}] \sqsubseteq o \text{.call } m(a)
\]

Similarly, for a semantics for references, if

\[
w, f : [\text{pre}, \text{post}[a \setminus f]] \sqsubseteq (o \uparrow o \circ m)[\text{self} \setminus o]
\]

with \( \text{pre} \) containing no \( f \) and neither \( f \) nor \( f_0 \) occurring in \( \text{post} \), then

\[
w, a : [\text{pre}, \text{post}] \sqsubseteq o \uparrow \text{.call } m(a)
\]

The nomenclature \textbf{result} is used to denote a result parameter.
B.4 Data Refinement Laws

To data refine guards, Gardiner and Morgan [GM91] introduce the following function.

\[ \overline{rep} \psi = \neg (rep \neg \psi) \]

When \( rep \) is of the form

\[ rep \psi \equiv (\exists a \bullet AI \land \psi) \]

then \( \overline{rep} \) is of the form

\[ \overline{rep} \psi \equiv (\forall a \bullet AI \Rightarrow \psi) \]

Using this function, guards data refine in the following manner.

Law B.63 (Data Refine Guard)

\[ [G] \preceq_{\overline{rep}} [\overline{rep} G] \]

B.5 Object-Refinement Laws

Theorem B.64 (Object-Refinement Reflexive) An object \( a \) is an object-refinement of itself.

\[ a \sqsubseteq a \]

Proof
The proof is inductive and relies on Definition 7.1, the reflexivity of equality (for the basic types) and refinement (for predicate transformers).

QED

Theorem B.65 (Object-Refinement Transitivity) When object \( a \) object-refines to \( b \), and \( b \) object-refines to \( c \), it can be deduced that \( a \) object-refines to \( c \).

\[ a \sqsubseteq b \quad b \sqsubseteq c \quad \therefore a \sqsubseteq c \]

Proof
The proof is inductive and relies on Definition 7.1, the transitivity of equality (for the basic types) and refinement (for predicate transformers).

QED
Theorem B.66 (Object-Data-Refinement Generalised Transitive)  Given objects $a$, $b$, $c$, rep $p = (\exists a \cdot AI \land p)$, and rep'$ p = (\exists a \cdot AI[b\backslash c] \land p)$ then:

$$a \preceq_{\text{rep}} b \land b \sqsubseteq^c c \Rightarrow a \preceq_{\text{rep}'} c$$

**Proof**

The proof is achieved by addressing each of the four conjuncts of Definition Object-Data-Refinement (7.39):

$$a \preceq_{\text{rep}'} c \equiv
\begin{align*}
\tau_{\text{Public}}(c) &\preceq \tau_{\text{Public}}(a) \land \\
\forall j \in \text{dom methods}(\tau_{\text{Public}}(a)) \Rightarrow a_{\circ j} \preceq_{\text{rep}'} c_{\circ j} \\
\forall j \in \text{dom fields}(\tau_{\text{Public}}(a)) \Rightarrow a_{\circ j} \sqsubseteq^c c_{\circ j} \land \\
&AI[b\backslash c]
\end{align*}$$

The first conjunct, $\tau_{\text{Public}}(c) \preceq \tau_{\text{Public}}(a)$, holds as:

$$a \preceq_{\text{rep}'} b \land b \sqsubseteq^c c \\
\Rightarrow
\begin{align*}
\tau_{\text{Public}}(b) &\preceq \tau_{\text{Public}}(a) \land \tau(c) \preceq \tau(b) \\
\Rightarrow
\tau_{\text{Public}}(c) &\preceq \tau_{\text{Public}}(a)
\end{align*}$$

The second conjunct, $\forall j \in \text{dom methods}(\tau_{\text{Public}}(a)) \Rightarrow a_{\circ j} \preceq_{\text{rep}'} c_{\circ j}$, holds as for each public method $j$:

$$a \preceq_{\text{rep}} b \land b \sqsubseteq^c c \\
\Rightarrow \text{Object-Data-Refinement (7.39)} \\
a_{\circ j} \preceq_{\text{rep}} c_{\circ j} \\
\equiv \text{Data Refinement (2.3)} \\
\text{rep}; \ a_{\circ j} \sqsubseteq b_{\circ j}; \ \text{rep} \\
\Rightarrow \text{Using the assumption } b \sqsubseteq^c c \text{ which entails } b_{\circ j} \sqsubseteq c_{\circ j} \\
\text{Refinement Monotonicity} \\
\text{Renaming variable } b \text{ to } c \\
\text{rep'}; \ a_{\circ j} \sqsubseteq c_{\circ j}; \ \text{rep'} \\
\equiv \text{Data Refinement (2.3)} \\
a_{\circ j} \preceq_{\text{rep}'} c_{\circ j}$$

The third conjunct, $\forall j \in \text{dom fields}(\tau_{\text{Public}}(a)) \Rightarrow a_{\circ j} \sqsubseteq^c c_{\circ j}$, holds given the following.

$$a \preceq_{\text{rep}} b \Rightarrow a_{\circ j} \sqsubseteq^c b_{\circ j}$$

Also,

$$b \sqsubseteq^c c \Rightarrow b_{\circ j} \sqsubseteq^c c_{\circ j}$$
Together,

\[ a \sqsubseteq j \sqsubseteq b \sqcap j \land b \sqsubseteq j \sqsubseteq c \sqcap j \Rightarrow a \sqsubseteq j \sqsubseteq c \sqsubseteq j \]

The final conjunct holds as \( AI \) is upwards closed.

\[ AI \land b \sqsubseteq c \Rightarrow AI[b \setminus c] \]

QED
Appendix C

Lattice Theoretic Semantics

Back and von Wright’s [BvW98] refinement calculus is founded in lattice and category theory, e.g., the refinement relation is a lattice ordering. This appendix provides an introduction to lattice theory.

Lattices are built from a number of other well known mathematical abstractions. One such abstraction is the partial order. A partial order is a relation that is reflexive, antisymmetric and transitive. Examples include \( \Rightarrow \) on the type \( \mathbb{B} \), \( \leq \) on the type \( \mathbb{N} \), and \( \subseteq \) on the powerset of any set \( S \), that is \( \mathbb{P}S \). An equivalence relation, in contrast, is reflexive, symmetric and transitive. Equality on the type \( \mathbb{N} \) is one example of an equivalence relation.

A partial order may have a bottom element. For example, the natural numbers, with \( \leq \) has zero as its bottom element. In the abstract, the bottom element is denoted as \( \bot \). The statement \texttt{abort} is the bottom of the refinement relation ordering in the refinement calculus. This means, amongst other things, that \texttt{abort} refines to anything, or alternatively, any program refines \texttt{abort}.

Besides the partial order relation, \( \leq \), a set may also have other operators between its members. For instance, the \texttt{min} function on the natural numbers takes two numbers and returns the greatest number that is less-than-or-equal to both numbers. This is called the meet or greatest lower bound. When not applied to a specific application, the meet is denoted by the symbol \( \sqcap \). The meet of booleans is boolean conjunction, the meet of sets (with \( \subseteq \) is intersection, the meet of predicates (with \( \Rightarrow \) is logical conjunction, and the meet of predicate transformers is demonic nondeterministic choice. The meet of a subset of a partial order’s set is the greatest lower bound or greatest element in the set of lower bounds of that subset. For example, the \texttt{min} function which is the meet for the poset \( (\mathbb{N}, \leq) \), when applied to the subset \( \{4, 6\} \), examines all lower bounds of the set, i.e., \( \{0, 1, 2, 3, 4\} \) and returns the greatest element, i.e., 4.

Joins are similarly defined, except that they return the least upper bound, rather than the greatest lower bound. That is, the least element of the set of upper bounds is returned by the join. For example, \( \sqcup \) is the join for the poset \( (\mathbb{P}S, \subseteq) \).
Lattices  Informally, a lattice is a partial order where the meet and join exists as a member of the lattice for any two elements. Example of lattices include $\Rightarrow$ on the type $\mathbb{B}$ with $\land$ as the meet and $\lor$ as the join, or alternatively, $\subseteq$ on the type $\mathbb{P} S$ with $\cap$ as the meet and $\cup$ as the join. The term lattice is derived from the lattice-like appearance that the graph of a lattice makes. Figure C.1 shows nine example lattices. For example, lattice four uses the set $\{a, b, c, d\}$ with the following relationships: $d \leq b$, $d \leq c$, $b \leq a$ and $c \leq a$. Lattice one is the only lattice that is possible with a single element. Lattice two is the only shape that a two member lattice can form. Similarly, lattice three is the only three member lattice shape. Lattices four and five are the four member lattices while the remainder are the five member lattices.
Appendix D
Proofs

This appendix provides the proofs for the theorems presented in this thesis. To aid the reader, the theorems have been duplicated inside boxes at the beginning of the corresponding proofs.

D.1 State and Predicate Proofs

Proof of 4.15 from p43 (Sub State Type)

Given state types $\alpha_{st}$ and $\beta_{st}$:

\[
\begin{align*}
\text{dom}(\beta) & \subseteq \text{dom}(\alpha) \\
\text{for all } (i \in \text{dom}(\beta)) \cdot \alpha(i) & \preceq \beta(i) \\
\text{for all } (i \in \text{dom}(\beta \preceq \alpha)) \cdot \alpha(i) & \preceq \beta_{st}
\end{align*}
\]

This property is consistent with that used by Sekerinski [Sek96].

Given respective state types $\alpha$ and $\beta$:

\[
\begin{align*}
\text{dom } \beta & \subseteq \text{dom } \alpha \\
\text{for all } i \in \text{dom } \beta \cdot \alpha(i) & \preceq \beta(i) \\
\text{for all } i \in \text{dom}(\beta \preceq \alpha) \cdot \alpha(i) & \preceq \beta_{st}
\end{align*}
\]

$\Rightarrow$ Sub Record (4.11)

$\alpha_{st} \preceq \beta_{st}$

$\Rightarrow$ States as Records (4.16)

$\alpha_{st} \preceq \beta_{st}$

QED
APPENDIX D. PROOFS

Proof of 4.18 from p44 (Sub Predicate)

Duplicate (Sub Predicate) of 4.18 on page 44.
The subtyping rule for predicates is

\[
\alpha \preceq \beta \\
\text{Pred } \beta \preceq \text{Pred } \alpha
\]

That is, predicates subtype when the states on which they are defined vary contravari-
nantly.

\[
\begin{align*}
\text{B} \\
\text{B} \preceq \text{B} & \quad \alpha \preceq \beta \\
\beta \rightarrow \text{B} \preceq \alpha \rightarrow \text{B} & \quad 4.10 \\
\text{Pred } \beta \preceq \text{Pred } \alpha & \quad \text{Def}^n 4.17 \\
\end{align*}
\]

QED

Proof of 4.20 from p45 (Val False)

Duplicate (Val False) of 4.20 on page 45.
For every well-formed state type \(\alpha\), \text{False} can be subsumed to a predicate on that state

type.

\[
\begin{align*}
\alpha \\
\text{False} : \text{Pred } \alpha
\end{align*}
\]

The predicate \text{True} can be constructed analogously.

\[
\begin{align*}
\text{False} : \text{Pred } \alpha \\
\iff \text{Sub Predicate (4.18), Val Subsumption (4.3)} \\
\text{False} : \text{Pred } \top_{st} \\
\alpha \preceq \top_{st} \\
\iff \text{Predicates (4.17), Sub Top (A.10)} \\
(\lambda s : \top_{st} \bullet \text{false}) : \top_{st} \rightarrow \text{B} \\
\text{Pred } \alpha \\
\iff \text{Assumption, Val Fun (A.5)} \\
s : \top_{st} \vdash \text{false} : \text{B}
\end{align*}
\]

QED

Proof of 4.22 from p45 (Val Predicate Implication)
\[(p_1 \Rightarrow p_2) : \text{Pred} (\alpha \sqcap \beta)\]

\(\Leftrightarrow\) Predicate Implication (4.21), provided \((\alpha \sqcap \beta)\) is well-formed.

\[\lambda s : (\alpha \sqcap \beta) . p_1(s) \Rightarrow p_2(s) : \text{Pred} (\alpha \sqcap \beta)\]

\(\Leftrightarrow\) Val Fun (A.5)

\[s : (\alpha \sqcap \beta) \vdash (p_1(s) \Rightarrow p_2(s)) : \mathbb{B}\]

\(\Leftrightarrow\) Val Appl (A.6)

\[p_1 : \text{Pred} (\alpha \sqcap \beta) \land p_2 : \text{Pred} (\alpha \sqcap \beta)\]

\(\Leftrightarrow\) Sub Predicate (4.18), Val Subsumption (4.3)

\[p_1 : \text{Pred} \alpha \land p_2 : \text{Pred} \beta\]

\textbf{QED}

\section*{D.2 Predicate Transformer Proofs}

\noindent \textbf{Proof of 4.26 from p46 (Sub Predicate Transformers)}

\begin{itemize}
  \item \textbf{Duplicate (Sub Predicate Transformers) of 4.26 on page 46.}
  \begin{itemize}
    \item Predicate transformers subtype when the postcondition state varies covariantly and the 
    precondition state varies contravariantly.
    \begin{align*}
      \alpha \preceq \beta & \quad \gamma \preceq \delta
    \end{align*}
    \begin{align*}
      Ptrans \alpha \delta & \preceq Ptrans \beta \gamma
    \end{align*}
    \end{itemize}
    as \text{Pred} \beta \preceq \text{Pred} \alpha \text{ and } \text{Pred} \delta \preceq \text{Pred} \gamma.
\end{itemize}

\noindent \textbf{QED}

\noindent \textbf{Proof of 4.29 from p48 (Val Abort)}

\begin{itemize}
  \item \textbf{Duplicate (Val Abort) of 4.29 on page 48.}
  \begin{itemize}
    \item Given state type \(\alpha\), the abort predicate transformer types as \(Ptrans \alpha \top_{str}\).
    \end{itemize}
\end{itemize}

\[\alpha \quad \text{abort}_\alpha : Ptrans \alpha \top_{str}\]
Proof of 4.31 from p48 (Val Demonic Choice)

Demonic Choice (4.30)
\[(\lambda p : \text{Pred } (\alpha \sqcup \beta) \bullet pt_1 p \land pt_2 p) : Ptrans (\alpha \sqcup \beta) (\delta \sqcap \gamma)\]
\[\equiv \text{Val Fun (A.5)}\]
\[p : \text{Pred } (\alpha \sqcup \beta) \vdash (pt_1 p \land pt_2 p) : \text{Pred } (\delta \sqcap \gamma)\]
\[\equiv \text{Val Conjunction (B.2)}\]
\[pt_1 p : \text{Pred } \delta\]
\[pt_2 p : \text{Pred } \gamma\]
\[(\delta \sqcap \gamma)\]

QED

D.3 Statement Proofs

This section provides explicit proofs of several basic properties of the simultaneous execution operator.

Theorem D.1 (Open Generalised Effect)  Given \(P : MT_{\alpha \rightarrow \psi}\) then
\[E_{\tilde{d},\tilde{w} \leftarrow \psi,\tilde{v},\tilde{w}}(P \oplus \tilde{w}) \equiv \neg P(\tilde{v} \neq \tilde{v}')[\tilde{w}, \tilde{w}', \tilde{v}, \tilde{v}' \setminus \tilde{w}_0, \tilde{w}, \tilde{v}_0, \tilde{v}]\]

Proof
APPENDIX D. PROOFS

Duplicate (Open Generalised Effect) of D.1 on page 196.
Given \( P : MT_{\bar{t} \rightarrow \bar{q}} \) then
\[
E_{\bar{t}, \bar{w} \rightarrow \bar{v}, \bar{w}}(P \oplus \bar{w}) \equiv \neg P(\bar{v} \neq \bar{v}')[\bar{w}, \bar{w}, \bar{v}, \bar{v}' \bar{w}_0, \bar{w}, \bar{v}_0, \bar{v}]
\]

Given \( P : MT_{\bar{t} \rightarrow \bar{q}} \) then
\[
E_{\bar{t}, \bar{w} \rightarrow \bar{v}, \bar{w}}(P \oplus \bar{w})
\]
(Notice \( \bar{v}\bar{w} \neq \bar{v}\bar{w}' \equiv \bar{v} \neq \bar{v} \lor \bar{w} \neq \bar{w}' \))
\[
\equiv \text{Defn w-opening (page 144).}
\]
\[
\neg (P; \bar{w} \rightarrow \bar{v}; [\text{True}] (\bar{v} \neq \bar{v} \lor \bar{w} \neq \bar{w}'))[\bar{w}, \bar{w}, \bar{v}, \bar{v}' \bar{w}_0, \bar{w}, \bar{v}_0, \bar{v}]
\]
\[
\equiv \text{Specification Statement (B.47)}
\]
\[
\neg (P (\forall \bar{w} \rightarrow \bar{v} \bullet (\bar{v} \neq \bar{v} \lor \bar{w} \neq \bar{w}'))[\bar{w}_0 - \bar{v}_0 - \bar{v}])[\bar{w}, \bar{w}, \bar{v}, \bar{v}' \bar{w}_0, \bar{w}, \bar{v}_0, \bar{v}]
\]
\[
\equiv \text{Partially Superfluous Universal Quantification (B.13)}
\]
\[
\neg (P (\bar{v} \neq \bar{v}' \lor \forall \bar{w} - \bar{v} \bullet \bar{w} \neq \bar{w}'))[\bar{w}, \bar{w}, \bar{v}, \bar{v}' \bar{w}_0, \bar{w}, \bar{v}_0, \bar{v}]
\]
\[
\equiv \text{Universal de Morgan (B.8)}
\]
\[
\neg (P (\bar{v} \neq \bar{v}' \lor \neg \exists \bar{w} - \bar{v} \bullet \bar{w} = \bar{w}'))[\bar{w}, \bar{w}, \bar{v}, \bar{v}' \bar{w}_0, \bar{w}, \bar{v}_0, \bar{v}]
\]
\[
\equiv \text{Existential Quantification One Point Rule (B.6)}
\]
\[
\neg (P (\bar{v} \neq \bar{v}' \lor \neg (\bar{v} = \bar{v}')))[\bar{w}, \bar{w}, \bar{v}, \bar{v}' \bar{w}_0, \bar{w}, \bar{v}_0, \bar{v}]
\]
\[
\equiv
\neg P(\bar{v} \neq \bar{v})[\bar{w}, \bar{w}, \bar{v}, \bar{v}' \bar{w}_0, \bar{w}, \bar{v}_0, \bar{v}]
\]
\[
QED
\]

Proof of 9.1 from p145 (Opened Generalised Effect Basic Properties)

Duplicate (Opened Generalised Effect Basic Properties) of 9.1 on page 145.
\[
E((x := e) \oplus \bar{w}) \equiv x = e[\bar{w}, \bar{x} \bar{w}_0, \bar{x}_0]
\]
\[
E(\bar{z}: [p, q] \oplus \bar{w}) \equiv p[\bar{z}, \bar{w}\bar{z}_0, \bar{w}_0] \Rightarrow q[\bar{w}\bar{w}_0]
\]

The proof is split into two parts: specifications and assignments.

Specifications:
\[
E(\bar{z}: [p, q] \oplus \bar{w})
\]
\[
\equiv \text{Open Generalised Effect (D.1)}
\]
\[
\neg (\bar{z}: [p, q] (\bar{z} \neq \bar{z}'))[\bar{z}, \bar{w}, \bar{z}', \bar{w}\bar{z}_0, \bar{w}_0, \bar{z}, \bar{w}]
\]
APPENDIX D. PROOFS

≡ Specification Statement (B.47)
≡ de Morgan’s Law

⇒ (p ∧ (∀ z • q ⇒ z ≠ z'))[z₀ \ z'][z; w, z', w₀ \ z₀, z; w]

≡ Definition of Implication

⇒ (q ⇒ (∃ z • ¬ (q ⇒ z ≠ z')))[z₀ \ z'][z; w, z', w₀ \ z₀, z; w]

≡ Applying Implication

⇒ (q ⇒ (p ∧ q ⇒ z ≠ z'))[z₀ \ z'][z; w, z', w₀ \ z₀, z; w]

≡ Applying Open Generalised Effect (D.1)

⇒ (e) ⊢ w

≡ Open Generalised Effect (D.1)

⇒ ((x := e')(x' ≠ x))[w; x, w', x₀ \ x₀, w, x]

≡ Assignment (B.30)

⇒ (x' ≠ x)[x \ e'][w; x, w', x₀ \ x₀, w, x]

≡ (x' = e)[w; x, w', x₀ \ x₀, w, x]

≡ Only x' free in x'

⇒ (x' ≠ e)[w; x, w', x₀ \ x₀, w, x]

≡ w', x' nf e

⇒ (x = e)[w; x \ x₀, x₀]

QED
D.4 Statement Refinements

Proof of 4.34 from p50 (Open World Specification)

Duplicate (Open World Specification) of 4.34 on page 50.
In an environment consisting only of state variables \( \tilde{z} \), and which is disjoint from \( \tilde{w} \):

\[
\tilde{z} : \left[ \text{pre} , \text{post} \right]_{\{\tilde{z} \}_{st}} \subseteq \tilde{z}, \tilde{w} : \left[ \text{pre} , \text{post} \right]_{\{\tilde{z}, \tilde{w} \}_{st}}
\]

Applying Definition Refinement (4.32) reduces this to the following. For all predicates \( p : \text{Pred} \{\tilde{z} : \tilde{Z}\}_{st} \):

\[
\left( \pre^p \right) \subseteq \left( \pre^p \right)_{\{\tilde{z} \}_{st}} \quad \Rightarrow \quad \left( \pre^p \right)_{\{\tilde{z}, \tilde{w} \}_{st}}
\]

\[
\equiv \text{Specification Statement (B.47)}
\]

\[
\pre \land (\forall \tilde{z} . \text{post} \Rightarrow p)[\tilde{z}_0 \setminus \tilde{z}]
\]

\[
\Rightarrow \quad \pre \land (\forall \tilde{z}, \tilde{w} . \text{post} \Rightarrow p)[\tilde{z}_0, \tilde{w}_0 \setminus \tilde{z}, \tilde{w}]
\]

\[
\equiv \text{Since } \tilde{w}_0 \text{ and } \tilde{w} \text{ are not free in either } p \text{ or post (post is a predicate on } \tilde{z}).
\]

\[
\pre \land (\forall \tilde{z} . \text{post} \Rightarrow p)[\tilde{z}_0 \setminus \tilde{z}]
\]

\[
\Rightarrow \quad \pre \land (\forall \tilde{z} . \text{post} \Rightarrow p)[\tilde{z}_0 \setminus \tilde{z}]
\]

QED

Proof of B.18 from p179 (Val Skip)

Duplicate (Val Skip) of B.18 on page 179.
Given state type \( \alpha \) the \( \text{skip}_\alpha \) predicate transformer types as

\[
\text{skip}_\alpha : Ptrans \alpha \alpha
\]

The proof uses Skip (B.17), Val Fun (A.5) and Val x (A.3).

QED

Proof of B.20 from p179 (Val Assertion)
Duplicate (Val Assertion) of B.20 on page 179.
The assertion \( \{p\}_\beta \) is typed as follows:

\[
p : \text{Pred} \quad (\alpha \sqcap \beta)
\]

\[
\{p\}_\beta : \text{Ptrans} \quad (\alpha \sqcap \beta)
\]

The state type \( \beta \) is chosen as required by the context. If \( \beta \) is chosen to be \( \alpha \) then the assertion types as

\[
\{p\}_\alpha : \text{Ptrans} \quad \alpha
\]

and the well-formedness of \( (\alpha \sqcap \alpha) \) reduces to the well-formedness of \( \alpha \).

The proof uses Assertion (B.19), Val Fun (A.5), Val Conjunction (B.2) and Val x (A.3).

QED

Proof of B.25 from p180 (Val Guard)

Duplicate (Val Guard) of B.25 on page 180.
The guard \( [p] \) is typed as follows:

\[
p : \text{Pred} \quad (\alpha \sqcap \beta)
\]

\[
[p]_\beta : \text{Ptrans} \quad (\alpha \sqcap \beta)
\]

The state type \( \beta \) is chosen as required by the context in an analogous manner to that for assertions.

The proof uses Guard (B.24), Val Fun (A.5), Val Predicate Implication (4.22) and Val x (A.3).

QED

Proof of B.29 from p181 (Val Update Statement)

Duplicate (Val Update Statement) of B.29 on page 181.
Update types as a predicate transformer from the range type of the state transformer function to its domain type.

\[
st : \alpha \rightarrow \beta
\]

\[
\langle st \rangle_\beta : \text{Ptrans} \quad \beta \quad \alpha
\]
The proof uses Update (B.28), Val Fun (A.5), and Val Appl (A.6).

**QED**

**Proof of B.31 from p181 (Val Assignment)**

**Duplicate (Val Assignment) of B.31 on page 181.**

Given state type $\alpha$ including distinct variables $i_j : I_{\in 1..m}$, and expressions $e_j : I_{\in 1..m}$ well defined in $\alpha$, an assignment statement types as follows:

$$(i_1, \ldots, i_m := e_1, \ldots, e_m)_{\alpha} : Ptrans \alpha \alpha$$

For Morgan’s definition, the assignment takes a predicate on state type $\alpha$ and also returns one as the variables $i_j \in 1..m$ have been syntactically replaced by expressions of the same type.

**QED**

**Proof of B.34 from p182 (Val Enter)**

**Duplicate (Val Enter) of B.34 on page 182.**

For state types ‘compatible’ with state type $\{v : V\}_{st}$, the enter predicate transformer types as follows:

$$v \not\in \text{dom}(\alpha) \quad \text{initv} : V$$

$$\text{enter } v : V := \text{initv})_{\alpha} : Ptrans (\alpha \sqcap \{v : V\}_{st}) \alpha$$

It is known that $(\alpha \sqcap \{v : V\}_{st})$ is well-formed from $v \not\in \text{dom}(\alpha)$.

**QED**

**Proof of B.36 from p182 (Val Exit)**

**Duplicate (Val Exit) of B.36 on page 182.**

Given state type $\alpha$,

$$v \not\in \text{dom}(\alpha)$$

$$(\text{exit } v : V)_\alpha \triangleq : Ptrans \alpha (\alpha \sqcap \{v : V\}_{st})$$

The proof uses Exit (B.35) and Val Update (A.14).

**QED**

**Proof of B.39 from p183 (Val Sequential Composition)**
### Duplicate (Val Sequential Composition) of B.39 on page 183.

Sequential composition, when defined, types as a predicate transformer from the post-condition state type of ‘second’ predicate transformer to the precondition state type of the ‘first’.

\[
pt_1 : P\text{trans }\alpha \delta \quad pt_2 : P\text{trans }\beta \gamma \quad \alpha \leq \gamma \\
(pt_1; pt_2) : P\text{trans }\beta \delta
\]

The proof uses Sequential Composition (B.38), Val Appl (A.6) and Sub Predicate (4.18).

**QED**

### Proof of B.42 from p184 (Val Local Variable Block)

Duplicate (Val Local Variable Block) of B.42 on page 184.

Given predicate transformer \( P : P\text{trans }\alpha \cap \{ a : A \}_{st} (\alpha \cap \{ a : A \}_{st}) \),

\[ |[\text{var } a : A = \text{av } \cdot P] |_{\alpha} : P\text{trans }\alpha \alpha \]

The proof uses Val Enter (B.34) and Val Exit (B.36).

For Morgan’s definition, \( \text{post} : \text{Pred } \alpha \), not containing \( a \), and the quantification removes \( a \).

**QED**

### Proof of B.44 from p184 (Val Logical Constant)

Duplicate (Val Logical Constant) of B.44 on page 184.

For state type \( \alpha \), and \( pt : P\text{trans }\alpha \alpha \),

\[ |[\text{con } \text{lcon } \cdot pt] |_{\alpha} : P\text{trans }\alpha \alpha \]

provided \( \text{lcon} \) is not free in \( \text{post} \).

The proof uses Logical Constant (B.43), Val Appl (A.6) and Val Fun (A.5).

**QED**

### Proof of B.48 from p185 (Val Specification)
Duplicate (Val Specification) of B.48 on page 185.
For state type $\alpha$, $v \in \text{dom}(\alpha)$, $\text{pre} : \text{Pred} \alpha$, and $\text{post} : \text{Pred} \alpha$,

$$v : \left[ \text{pre} \cdot \text{post} \right]_\alpha : \text{Ptrans} \alpha \alpha$$

The proof uses Specification Statement (B.47), Val Fun (A.5), Val Appl (A.6), Val Conjunction (B.2) and Val Predicate Implication (4.22).

QED

Proof of B.57 from p186 (Val Alternation)

Duplicate (Val Alternation) of B.57 on page 186. Given guards $G_{i \in 1..n} : \text{Pred} \alpha$ and predicate transformers $P_{i \in 1..n} : \text{Ptrans} \alpha \alpha$:

$$\text{if } [G_{i \in 1..n} \rightarrow P_i] \text{ fi } : \text{Ptrans} \alpha \alpha$$

The proof uses Alternation Statement (B.56), Val Fun (A.5), Val Appl (A.6), Val Conjunction (B.2), Val Disjunction (B.4), and Val Predicate Implication (4.22).

QED

Proof of B.60 from p187 (Val Iteration)

Duplicate (Val Iteration) of B.60 on page 187. Given guards $G_{i \in 1..n} : \text{Pred} \alpha$, and $\text{prog}_{i \in 1..n} : \text{Ptrans} \alpha \alpha$:

$$\text{do } ( [G_{i \in 1..n} \rightarrow \text{prog}_i] ) \text{ od } : \text{Ptrans} \alpha \alpha$$

Given $\text{prog}_{i \in 1..n} : \text{Ptrans} \alpha \alpha$ and assuming $p : \text{Ptrans} \alpha \alpha$ then

$$\text{if } [G_{i \in 1..n} \rightarrow \text{prog}_i] \text{ fi } : \text{Ptrans} \alpha \alpha$$

The type of $\text{re } p \circ C(p) \text{ er}$ is the type of $C$ as $\text{re } p \circ C(p) \text{ er}$ is defined such that it is the least-refined program $p$ where $C(p) = p$. Hence the type of

$$\text{re } p \circ \text{if } [G_{i \in 1..n} \rightarrow \text{prog}_i] \text{ fi } : \text{Ptrans} \alpha \alpha$$

is $\text{Ptrans} \alpha \alpha$.

QED

Proof of B.21 from p180 (Weaken Assertion)
Duplicate (Weaken Assertion) of B.21 on page 180.

Given predicates \( p : \perp \) \( \text{Pred} \ P \) and \( q : \text{Pred} \ Q \), and state type \( \alpha \) such that 
\[
(P \text{Pred} P \sqcap (P \text{Pred} Q \sqcap \alpha)),
\]

\[
p \Rightarrow q
\]

\[
\{p\}_\alpha \subseteq \{q\}_\alpha
\]

For any postcondition predicate \( \text{post} : \text{Pred} \ \alpha \).

\[
\{p\}_\alpha \text{ post}
\]
\( \equiv \) Assertion (B.19)

\[
(\lambda r : \text{Pred} \ \alpha \bullet p \land r) \text{ post}
\]
\( \equiv \) Function Application

\[
p \land \text{post}
\]
\( \Rightarrow \) Antecedent

\[
q \land \text{post}
\]
\( \equiv \) Function Abstraction

\[
(\lambda r : \text{Pred} \ \alpha \bullet q \land r) \text{ post}
\]
\( \equiv \) Assertion (B.19)

\[
\{q\}_\alpha \text{ post}
\]

The proof is completed using Refinement (4.32).

\( QED \)
D.5 Client Constructs Pre Object-Refinement

Proof of 6.6 from p67 (Expanding the Frame with a Postfix Closure)

Duplicate (Expanding the Frame with a Postfix Closure) of 6.6 on page 67.
The object specification frame may be extended with an attribute path \((p)\) that is in the postfix closure of an attribute path \((o)\) that is already in the frame.

\[
o:: [\text{post}] \equiv o.p:: [\text{post}]
\]

provided \(p \in \text{poc}(o)\).

\[
\begin{align*}
o:: [\text{post}] \\
\equiv & \text{Object Specification (Semantics for Values) (6.5)} \\
\text{head}(o): & \left[ \bigwedge a \in \left( \bigcup \left\{ s \in \text{head}(o) \cdot \text{poc}(s) \right\} \setminus \bigcup \left\{ s \in o \cdot \text{poc}(s) \cup \text{prec}(s) \right\} \right) \cdot a = a_0 \right] \\
\equiv & \text{If } p \in \text{poc}(o) \text{ then } \text{head}(o) \equiv \text{head}(o \cup p) \\
\text{head}(o \cup p): & \left[ \bigwedge a \in \left( \bigcup \left\{ s \in \text{head}(o \cup p) \cdot \text{poc}(s) \right\} \setminus \bigcup \left\{ s \in o \cdot \text{poc}(s) \cup \text{prec}(s) \right\} \right) \cdot a = a_0 \right] \\
\equiv & \text{If } p \in \text{poc}(o) \text{ then } \\
\text{prec}(p) & \subseteq (\text{poc}(o) \cup \text{prec}(o)) \land \text{poc}(p) \subseteq \text{poc}(o) \\
\text{and} & \\
\bigcup \left\{ s \in (o \cup p) \cdot \text{poc}(s) \cup \text{prec}(s) \right\} \\
\equiv & \text{Splitting off } p. \\
\bigcup \left\{ s \in o \cdot \text{poc}(s) \cup \text{prec}(s) \right\} \cup \text{poc}(p) \cup \text{prec}(p) \\
\text{head}(o \cup p): & \left[ \bigwedge a \in \left( \bigcup \left\{ s \in \text{head}(o \cup p) \cdot \text{poc}(s) \right\} \setminus \bigcup \left\{ s \in o \cup p \cdot \text{poc}(s) \cup \text{prec}(s) \right\} \right) \cdot a = a_0 \right] \\
\equiv & \text{Object Specification (Semantics for Values) (6.5)} \\
o.p:: [\text{post}] \\
\end{align*}
\]

QED

Theorem D.2 (Expanding the Frame with a Postfix Closure Reproof) Proof on page 206 Refer to page 87 for a discussion of the distinction between this theorem and that of the previous proof. The object specification frame may be extended with an attribute path \((p)\) that is in the postfix closure of an attribute path \((o)\) that is already in the frame.

\[
o:: [\text{post}] \equiv o.p:: [\text{post}]
\]

provided \(p \in \text{poc}(o)\). This theorem supports the maintenance of object-refinement monotonicity.
Proof of D.2 from p205

The proof is almost identical to that of Expanding the Frame with a Postfix Closure (6.6). The only differences are that the equality relation is replaced by object-refinement relation and Definition Modified Object Specification (Semantics for Values) (7.14) is used instead of Object Specification (Semantics for Values) (6.5).

QED

Theorem D.3 (Generalised Extending the Frame with a Reference Closure) For object specifications using reference attribute paths, base attribute paths $B$, dereference attribute paths $D$, reference attribute paths $R$, and reference closure attribute paths $C$,

$$B, D, R, C, o!:: [post] \equiv B, D, R, C, p, o!:: [post]$$

provided ${\text{front}(p) \mapsto \text{last}(p)} \in \text{refcl}(o)$.

Proof

$$\zeta(D, R \cup \{p\}, C \cup \{o\})$$

$\equiv$ Definition of $\zeta$.

$$\{i \in D \mapsto \text{attributes}(i)\} \uplus \{i \in R \cup \{p\} \mapsto \text{front}(i) \mapsto \text{last}(i)\} \uplus \text{refcl}(\| C \cup \{o\})$$

$\equiv$

$$\{i \in D \mapsto \text{attributes}(i)\} \uplus \{i \in R \mapsto \text{front}(i) \mapsto \text{last}(i)\} \uplus \{\text{front}(p) \mapsto \text{last}(p)\} \uplus \text{refcl}(\| C \cup \{o\})$$

$\equiv$ Assumption: $\{\text{front}(p) \mapsto \text{last}(p)\} \in \text{refcl}(o)$

$\zeta(D, R, C \cup \{o\})$

The proof is completed using Definition Object Specification (Semantics for References) (6.16).

QED

Proof of 6.17 from p76 (Extending the Frame with a Reference Closure)

Duplicate (Extending the Frame with a Reference Closure) of 6.17 on page 76. For object specifications using reference attribute paths, the frame can be extended with a reference ($p$) in the reference closure of a reference closure attribute path ($o!$).

$$o!:: [post] \equiv p, o!:: [post]$$

provided $\{\text{front}(p) \mapsto \text{last}(p)\} \in \text{refcl}(o)$.

The proof is an application of Generalised Extending the Frame with a Reference Closure (D.3) using $B \equiv \emptyset$, $D \equiv \emptyset$, and $R \equiv \emptyset$.

QED
Theorem D.4 (Extending the Frame with a Reference Closure Reproof)  For object specifications using reference attribute paths, the frame can be extended with a reference \( (p) \) in the reference closure of a reference closure attribute path \( (o!) \).

\[
o!: \[ post \] \equiv p, o!: \[ post \]
\]

provided \( \{ \text{front}(p) \rightarrow \text{last}(p) \} \in \text{refcl}(o) \). This theorem supports the maintenance of object-refinement monotonicity.

Proof
The proof is identical to that of Extending the Frame with a Reference Closure (6.17).

\( \text{QED} \)

Proof of 6.9 from p68 (Introduce Field Update (Semantics for Values) A)

Duplicate (Introduce Field Update (Semantics for Values) A) of 6.9 on page 68.

Given object value variable \( o \) and an expression \( e \) which may not include initial variables:

\[
o.f :: \[ o.f = e[o\backslash o_0] \] \equiv o.f := e
\]

\[
o.f :: \[ o.f = e[o\backslash o_0] \]
\equiv \text{Object Specification (Semantics for Values) (6.5)}
\equiv \text{Eval Update (4.9)}
\equiv \text{Simple Specification (B.52)}
\equiv \text{Object Field Update (Semantics for Values) (6.8)}
\]

\( \text{QED} \)

Proof of 6.19 from p76 (Introduce Field Update (Semantics for References) A)

Duplicate (Introduce Field Update (Semantics for References) A) of 6.19 on page 76.

Assume an object reference \( o \) and expression \( e \). An object specification where the attribute \( f \) of \( o \) may be altered such that its final value is \( e \) is equivalent to updating field \( f \) of \( o \) with \( e \):

\[
o^\uparrow f :: \[ o^\uparrow f = e_0 \] \equiv o^\uparrow f := e
\]

where \( e_0 \) is \( e \) with all dereferences of \( \text{store} \) replaced by corresponding dereferences of \( \text{store}_0 \).
$o \uparrow f :: [ o \uparrow f = e_0 ]$

≡ Object Specification (Semantics for References) (6.16)

\[ o \uparrow f = e_0 \land \]
\[ \forall n \in \text{dom}(\text{store}) \setminus \{ o \} \cdot \text{store}(n) = \text{store}_0(n) \land \]
\[ \forall m \in \text{attributes}(\text{store}(o)) \setminus \{ f \} \cdot \]
\[ \text{store}(o).m = \text{store}_0(o).m \]

≡ Object Field Selection (6.2)

\[ \text{store}(o) :: f = e_0 \land \]
\[ \forall n \in \text{dom}(\text{store}) \setminus \{ o \} \cdot \text{store}(n) = \text{store}_0(n) \land \]
\[ \forall m \in \text{attributes}(\text{store}(o)) \setminus \{ f \} \cdot \text{store}(o).m = \text{store}_0(o).m \]

≡ Eval Select (4.6)

\[ \text{store}(o) = \text{store}_0(o) :: f \rightleftharpoons e_0 \land \]
\[ \forall n \in \text{dom}(\text{store}) \setminus \{ o \} \cdot \text{store}(n) = \text{store}_0(n) \]

≡ Property of function override

\[ \text{store} :: [ \text{store} = \text{store}_0 \oplus \{ o \mapsto \text{store}_0(o) :: f \rightleftharpoons e_0 \} ] \]

≡ Simple Specification (B.52)

\[ \text{store} := \text{store} \oplus \{ o \mapsto \text{store}(o) :: f \rightleftharpoons e \} \]

≡ Accessed Function Assignment (6.12)

\[ \text{store}(o) := \text{store}(o) :: f \rightleftharpoons e \]

≡ Field Update (Semantics for References) (6.18)

\[ o \uparrow f := e \]

QED

### D.6 Object-Refinement Proofs

**Proof of 7.2 from p80 (Update Object Field)**

**Duplicate (Update Object Field) of 7.2 on page 80.**

Replacing an object’s field with an object-refinement is an object-refinement. If object $o$ has field $l$ and $o \circ l \sqsubseteq \simeq f$, then $o$’s $l$ field can be replaced by $f$.

\[
\frac{
    o \circ l \sqsubseteq \simeq f
}{
    o \sqsubseteq \simeq (o \circ l \rightleftharpoons f)
}
\]

The construct $o \circ l \rightleftharpoons f$ (introduced in Section 4.1) is object calculus syntax representing the replacement of $o$’s field $l$ with $f$.

Straightforward application of Definition Object-Refinement Algorithmic (7.1) and Eval Update (4.9).

QED
Proof of 7.4 from p81 (Update Object Field Refined)

**Duplicate (Update Object Field Refined) of 7.4 on page 81.**
If e object-refines to f, then updating any field of any object o with e object-refines to updating o with f instead.

\[
e \sqsubseteq f \\
\implies (o \circ fld \leftarrow e) \sqsubseteq (o \circ fld \leftarrow f)
\]

\[
e \sqsubseteq f \\
\equiv \text{Eval Select (4.6)} \\
(\circ fld \leftarrow e) \circ fld \sqsubseteq f \\
\Rightarrow \text{Update Object Field (7.2)} \\
(\circ fld \leftarrow e) \sqsubseteq ((\circ fld \leftarrow e) \circ fld \leftarrow f) \\
\equiv \text{Eval Update (4.9)} \\
(\circ fld \leftarrow e) \sqsubseteq (\circ fld \leftarrow f)
\]

QED

Proof of 7.5 from p81 (Update Object Method)

**Duplicate (Update Object Method) of 7.5 on page 81.**
This refinement rule permits the refinement of an object’s method, resulting in an object-refinement. Given an object o with a method l, and a method m that is a refinement of o\circ l, then the replacement of o\circ l with m is a valid object-refinement.

\[
o \circ l \sqsubseteq m \\
o \sqsubseteq (o \circ l \leftarrow m)
\]

Straightforward application of Definition Object-Refinement Algorithmic (7.1), and Eval Update (4.9).

QED

Proof of 7.7 from p82 (Introduce Object Attributes)
Duplicate (Introduce Object Attributes) of 7.7 on page 82.

Given an object with fields $f_{1..j}$ and methods $m_{1..k}$, adding new fields $f_{i+1..i+j}$ (for $j \geq 0$) and methods $m_{k+1..k+p}$ (for $p \geq 0$) to an object produces an object-refinement. For field values $fv_{1..i+j}$ of types $F_{1..i+j}$ and methods $mv_{1..k+p}$:

```plaintext
object
  field $f_{h \in 1..i}$ : $F_h := fv_h$
  method $m_{h \in 1..k} = mv_h$
end

```

Straightforward application of Definition Object-Refinement Algorithmic (7.1), the reflexivity of refinement and Object-Refinement Reflexive (B.64).

QED
D.7 Client Constructs - Object-Refinement Proofs

Proof of 7.11 from p86 (Object-Refine in Assignment (Semantics for Values))

Duplicate (Object-Refine in Assignment (Semantics for Values)) of 7.11 on page 86.

For variable $o$ and expressions $e$ and $f$:

$$ e \sqsubseteq^* f $$

$$(o := e) \sqsubseteq (o := f)$$

That is, substituting $e$ for $f$ in the assignment $o := e$ produces the refined statement $o := f$.

Straightforward application of the definitions of refinement, assignment and the monotonicity of predicates.

\[
\begin{align*}
RHS & \\
\equiv & \text{Refinement (4.32)} \\
\forall A \bullet ((o := e) A) \Rightarrow ((o := f) A) \\
\equiv & \text{Assignment (B.30)} \\
\forall A \bullet A[o/e] \Rightarrow A[o/f] \\
\sqsubseteq & \text{Upwards closure of } A \\
e & \sqsubseteq^* f
\end{align*}
\]

QED

Proof of 7.12 from p86 (Object-Refine in Specification (Semantics for Values))

Duplicate (Object-Refine in Specification (Semantics for Values)) of 7.12 on page 86.

For variable $o$ and expressions $e$ and $f$:

$$ e \sqsubseteq^* f $$

$$ o: [e \sqsubseteq^* o] \sqsubseteq o: [f \sqsubseteq^* o] $$

Straightforward application of Strengthen Postcondition (B.54) and Object-Refinement Transitivity (B.65).

QED

Proof of 7.13 from p86 (Object-Refinement Specification)
Duplicate (Object-Refinement Specification) of 7.13 on page 86.

For an expression $e$ that does not contain any initial variables:

\[ o : [ e[o \backslash o_0] \sqsubseteq o ] \equiv o := e \]

**LHS(A)**

\[ o : [ e[o \backslash o_0] \sqsubseteq o ] A \]

$\equiv$ Specification Statement (B.47)

\[ (\forall o \bullet e[o \backslash o_0] \sqsubseteq o \Rightarrow A[o_0 \backslash o]) \]

$\equiv$ Rename bound variable $o$ to $o'$

\[ (\forall o' \bullet e[o \backslash o_0] \sqsubseteq o' \Rightarrow A[o \backslash o'])[o_0 \backslash o] \]

$\equiv$ $o_0$ is not free in $e$ and $o_0$ is not free in $A$

\[ (\forall o' \bullet e \sqsubseteq o' \Rightarrow A[o \backslash o']) \]

**RHS(A)**

\[ (o := e) A \]

$\equiv$ Assignment (B.30)

\[ A[o \backslash e] \]

The proof consequently reduces to showing that

\[ (\forall o' \bullet e \sqsubseteq o' \Rightarrow A[o \backslash o']) \equiv A[o \backslash e] \]

This is achieved by showing entailment in both directions.

**Forwards:**

\[ (\forall o' \bullet e \sqsubseteq o' \Rightarrow A[o \backslash o']) \]

$\Rightarrow$ Universal Quantification Elimination (B.15)

\[ e \sqsubseteq e \Rightarrow A[o \backslash e] \]

$\equiv$ Object-Refinement Reflexive (B.64)

\[ A[o \backslash e] \]

**Backwards:**

\[ true \]

$\equiv A$ is upwards closed.

\[ (\forall o' \bullet e \sqsubseteq o' \Rightarrow (A[o \backslash e] \Rightarrow A[o \backslash o'])) \]

$\Rightarrow o'$ is not free in $A[o \backslash e]$

\[ A[o \backslash e] \Rightarrow (\forall o' \bullet e \sqsubseteq o' \Rightarrow A[o \backslash o']) \]

QED

Proof of 7.16 from p87 (Expand Frame New Fields (Semantics for Values))
In a semantics for values, given $n \geq 0, j \geq 0, 0 \leq k \leq n, p \geq 0, r \geq j + 1$, frames $frame_{i \in 1..n}$ that are each subsets of the fields $f_{i \in 1..n}$:

$$\forall i \in 1..j \cdot frame_i \subseteq flds$$

where

$$flds \equiv \bigcup_{i=1}^{n} \{self.f_i\}$$

If it is also assumed that

$$o \equiv \text{object}$$

$$\text{field } f_{i \in 1..n} : F_i := fv_i$$

$$\text{method } m_{i \in 1..j} = frame_i :: [q]$$

$$\text{method } m_{i \in j+1..r} = mv_i$$

end

and

$$o' \equiv \text{object}$$

$$\text{field } f_{i \in 1..n+p} : F_i := fv_i$$

$$\text{method } m_{i \in 1..j} = frame_i, self.f_{n+1..n+p} :: [q]$$

$$\text{method } m_{i \in j+1..r} = mv_i$$

end

then $o \sqsubseteq o'$.

Assume the shorthand $nfm$ for the fields not in the frame $frame$:

$$nfm(frame) \equiv \{self.findex \in (flds \setminus frame) \cdot findex\}$$

Using Definition 7.1, Sub Refl (A.7), Object-Refinement Reflexive (B.64) and the reflexivity of refinement, the proof reduces to showing for each $i \in 1..j$ that $o.m_i \sqsubseteq o'.m_i$. For each $m_{i \in 1..j}$:

$$frame :: [q]_{\{self\.\tau(o)\}}_{ST} \sqsubseteq frame_i, self.f_{i\in 1..n+p} :: [q]_{\{self\.\tau(o')\}}_{ST}$$

$\sqsubseteq$ Modified Object Specification (Semantics for Values) (7.14)

$$self : [q \wedge \forall fld \in nfm(frame_i) \cdot self.fld \sqsupseteq self_0.fld]_{\{self\.\tau(o)\}}_{ST} \sqsubseteq$$

$$self : [q \wedge \forall fld \in nfm(frame_i) \cdot self.fld \sqsupseteq self_0.fld]_{\{self\.\tau(o')\}}_{ST}$$

The proof is completed using Refinement (4.32) and Specification Statement (B.47).

QED

Proof of 7.17 from p88 (Expand Frame New Fields (Semantics for References))
APPENDIX D. PROOFS

Duplicate (Expand Frame New Fields (Semantics for References)) of 7.17 on page 88.

In a semantics for references, given $n \geq 0, j \geq 0, 0 \leq k \leq n, p \geq 0$,

$$o \equiv \text{object}$$

\[ \begin{align*}
& \text{field } f_{i \in 1..n} : F_i := f v_i \\
& \text{method } m_{i \in 1..j} = m v_i \\
& \text{method } \text{meth} = \text{self} \uparrow f_{1..k} : [q]
\end{align*} \]

end

and

$$o' \equiv \text{object}$$

\[ \begin{align*}
& \text{field } f_{i \in 1..n+p} : F_i := f v_i \\
& \text{method } m_{i \in 1..j} = m v_i \\
& \text{method } \text{meth} = \text{self} \uparrow f_{1..k}, \text{self} \uparrow f_{n+1..n+p} : [q]
\end{align*} \]

end

then $o \subseteq o'$.

Assume the shorthand $nfm$ for the fields not in the frame $frame$:

$$nfm(frame) \equiv \{ \text{self} \uparrow f_{\text{index}} \in (\text{flds} \setminus frame) \bullet f_{\text{index}} \}$$

Using Definition 7.1, Sub Refl (A.7), Object-Refinement Reflexive (B.64) and the reflexivity of refinement, the proof reduces to showing for each $i \in 1..j$ that $o \uparrow . m_i \subseteq o' \uparrow . m_i \in 1..j$.

For $m_i$:

$$frame \uparrow [q] \{ \text{self} \uparrow \tau(a) \}_{n_{f_{\text{index}}} \text{str}} \subseteq frame_i, \text{self} \uparrow f_{n+1..n+p} : [q] \{ \text{self} \uparrow \tau(a') \}_{n_{f_{\text{index}}} \text{str}}$$

$\equiv$ Modified Object Specification (Semantics for References) (7.15)

\[ \begin{align*}
\text{store} : \begin{cases}
q \land \\
& \forall n \in \text{dom}(\text{store}) \setminus \{ \text{self} \} \bullet \text{store}_0(n) \subseteq \text{store}(n) \\
& \forall a \in nfm(frame_i) \bullet \\
& \text{self} \uparrow 0.a \subseteq \text{self} \uparrow a \\
\end{cases} \subseteq \\
\text{store} : \begin{cases}
q \land \\
& \forall n \in \text{dom}(\text{store}) \setminus \{ \text{self} \} \bullet \text{store}_0(n) \subseteq \text{store}(n) \\
& \forall a \in nfm(frame_i) \bullet \\
& \text{self} \uparrow 0.a \subseteq \text{self} \uparrow a \\
\end{cases} \{ \text{self} \uparrow \tau(a) \}_{n_{f_{\text{index}}} \text{str}} \\
\text{store} : \begin{cases}
q \land \\
& \forall n \in \text{dom}(\text{store}) \setminus \{ \text{self} \} \bullet \text{store}_0(n) \subseteq \text{store}(n) \\
& \forall a \in nfm(frame_i) \bullet \\
& \text{self} \uparrow 0.a \subseteq \text{self} \uparrow a \\
\end{cases} \{ \text{self} \uparrow \tau(a') \}_{n_{f_{\text{index}}} \text{str}}
\end{align*} \]

The proof is completed using Refinement (4.32) and Specification Statement (B.47).

QED

Proof of 7.18 from p88 (Object Specification Weaken Precondition)
Duplicate (Object Specification Weaken Precondition) of 7.18 on page 88.
The precondition of an object specification can be weakened. Given \( \text{pre} \vdash \text{pre}' \):

\[
L :: [\text{pre, post}] \subseteq L :: [\text{pre}', \text{post}]
\]

The proof for a value semantics and a reference semantics is the same: use Dual Predicate Object Specification (6.7), Weaken Assertion (B.21), and finally, Dual Predicate Object Specification (6.7) again.

\( QED \)

Proof of 7.19 from p89 (Object Specification Strengthen Postcondition)

Duplicate (Object Specification Strengthen Postcondition) of 7.19 on page 89.
The postcondition of an object specification can be strengthened. Given

\[
\text{pre}[\text{head}(j L j) \wedge \text{pre}'] \vdash \text{post}
\]

then

\[
L :: [\text{pre, post}] \subseteq L :: [\text{pre}', \text{post}']
\]

The proofs for a value semantics and a reference semantics are similar: use Dual Predicate Object Specification (6.7), Modified Object Specification (Semantics for Values) (7.14) (or Modified Object Specification (Semantics for References) (7.15) as appropriate), Strengthen Postcondition (B.54), Object Specification (Semantics for Values) (6.5) (or Object Specification (Semantics for References) (6.16)), and finally, Dual Predicate Object Specification (6.7).

\( QED \)

Proof of 7.20 from p89 (Object Specification Contract Frame)

Duplicate (Object Specification Contract Frame) of 7.20 on page 89.
The frame of an object specification can be contracted. Given an object with fields \( f \) and \( g \):

\[
o.f, o.g :: [\text{pre, post}] \subseteq o.f :: [\text{pre, post}]
\]
The proof assumes $o$ has fields $f$, $g$, and $h$:

\[ o.f, o.g:: [pre, post] \]
\[ \equiv \text{Dual Predicate Object Specification (6.7)} \]
\[ \quad \text{Modified Object Specification (Semantics for Values) (7.14)} \]
\[ \{pre\}; \]
\[ o:: [post \land o_0.h \subseteq^c o.h] \]
\[ \equiv \text{Strengthen Postcondition (B.54)} \]
\[ \{pre\}; \]
\[ o:: [post \land o_0.h \subseteq^c o.h \land o_0.g \subseteq^c o.g] \]
\[ \equiv \text{Dual Predicate Object Specification (6.7)} \]
\[ \quad \text{Object Specification (Semantics for Values) (6.5)} \]
\[ o.f:: [pre, post] \]

QED

**Proof of 7.21 from p89 (Introduce Sequential Composition (Semantics for Values))**

**Duplicate (Introduce Sequential Composition (Semantics for Values)) of 7.21 on page 89.**

An object specification can be refined to a sequential composition. For an object with fields $f$, $g$ and $h$:

\[ o.f, o.g:: [q] \]
\[ \equiv \]
\[ ||\text{con } O \bullet \]
\[ o.f:: [p]; o.f, o.g:: [p[o_0\backslash O] \land o.h \supseteq^c O.h \land o.g \supseteq^c O.g \land q[o_0\backslash O]] \]

\[ o.f, o.g:: [q] \]
\[ \equiv \text{Modified Object Specification (Semantics for Values) (7.14)} \]
\[ o:: [q \land o.h \supseteq^c o_0.h] \]
\[ \equiv \text{Introduce Sequential Composition (B.40)} \]
\[ \quad \text{Strengthen Postcondition (B.54)} \]
\[ ||\text{con } O \bullet \]
\[ o:: [p \land o.h \supseteq^c o_0.h \land o.g \supseteq^c o_0.g]; \]
\[ o:: [p[o_0\backslash O] \land o.h \supseteq^c O.h \land o.g \supseteq^c O.g \land q[o_0\backslash O] \land o.h \supseteq^c O.h] \]
Proof of 7.22 from p89 (Object Specification Alternation)

**Duplicate (Object Specification Alternation) of 7.22 on page 89.**

An object specification can be refined to an alternation. Given \( \text{pre} \Rightarrow \text{GG} \) where \( \text{GG} \) is the disjunction of the guards \( G_{i \in 1..n} \):

\[
\text{w:: [pre , post]} \sqsubseteq \text{if } \bigwedge_{i \in 1..n} G_i \rightarrow \text{w:: [pre \land G_i , post]} \quad \text{fi}
\]

The proof uses Modified Object Specification (Semantics for Values) (7.14), Alternation (B.58), under the assumption \( \text{pre} \Rightarrow \text{GG} \), and Modified Object Specification (Semantics for Values) (7.14) again.

\( QED \)

**Proof of 7.23 from p89 (Object Specification Iteration)**
Duplicate (Object Specification Iteration) of 7.23 on page 89.

An object specification can be refined to an iteration. Given an object \( o \) with fields \( f \) and \( g \) (\( g \) is used only in the proof), the invariant \( \text{inv} \), the integer variant \( V \) and the disjunction of the guards \( GG \):

\[
\begin{align*}
o.f:: [ \text{inv} \land inv \land \lnot GG ] \\
\equiv \\
do ( \{ \forall i \in \{1..n\} G_i \rightarrow o.f:: [ \text{inv} \land G_i \land (0 \leq V < V_0) ] \} ) \text{ od}
\end{align*}
\]

Neither \( \text{inv} \) or \( GG \) may contain initial variables.

\[
\begin{align*}
o.f:: [ \text{inv} \land inv \land \lnot GG ] \\
\equiv & \text{ Modified Object Specification (Semantics for Values) (7.14)} \\
o:: [ \text{inv} \land inv \land \lnot GG \land o.g \supseteq o_0.g ] \\
\equiv & \text{ Fix Initial Value (B.46)} \\
\{ \text{con } OG \bullet \\
o:: [ \text{inv} \land o.g \supseteq OG \land inv \land \lnot GG \land o.g \supseteq OG ] \\
\} \\
\equiv & \text{ Iteration (B.61)} \\
\text{Using the invariant } \text{inv} \land o.g \supseteq OG \\
\{ \text{con } OG \bullet \\
do \{ \forall i \in \{1..n\} G_i \rightarrow \\
o:: [ \text{inv} \land o.g \supseteq OG \land G_i \land inv \land o.g \supseteq OG \land (0 \leq V < V_0) ] \\
\text{ od} \\
\} \\
\equiv & \text{ Strengthen Postcondition (B.54)} \\
\text{Weaken Precondition (B.53)} \\
\{ \text{con } OG \bullet \\
do \{ \forall i \in \{1..n\} G_i \rightarrow \\
o:: [ \text{inv} \land G_i \land inv \land o.g \supseteq o_0.g \land (0 \leq V < V_0) ] \\
\text{ od} \\
\} \\
\equiv \text{ Modified Object Specification (Semantics for Values) (7.14)} \\
\{ \text{con } OG \bullet \\
do \{ \forall i \in \{1..n\} G_i \rightarrow \\
o.f:: [ \text{inv} \land G_i \land inv \land (0 \leq V < V_0) ] \\
\text{ od} \\
\} \\
\equiv & \text{ Remove Logical Constant (B.45)} \\
do \{ \forall i \in \{1..n\} G_i \rightarrow \\
o.f:: [ \text{inv} \land G_i \land inv \land (0 \leq V < V_0) ] \\
\text{ od} \\
QED
\]
Proof of 7.24 from p90 (Object-Refine in Assignment (Semantics for References))

Duplicate (Object-Refine in Assignment (Semantics for References)) of 7.24 on page 90.
Object-refining an object in a dereference assignment produces a refinement. For reference variable \( o \) and expressions \( e \) and \( f \):

\[
\begin{align*}
  e & \sqsubseteq^e f \\
  (o^\uparrow := e) & \sqsubseteq (o^\uparrow := f)
\end{align*}
\]

\[
\begin{align*}
  (o^\uparrow := e) & \sqsubseteq (o^\uparrow := f) \\
  \equiv & \quad \text{Refinement (4.32)} \\
  \forall B \cdot (o^\uparrow := e)B & \Rightarrow (o^\uparrow := f)B \\
  \equiv & \quad \text{Accessed Function Assignment (6.12), Object Dereference (6.13)} \\
  \forall B \cdot (store := store \oplus \{o \mapsto e\})B & \Rightarrow (store := store \oplus \{o \mapsto f\})B \\
  \equiv & \quad \text{Assignment (B.30)} \\
  \forall B \cdot B[store \backslash store \oplus \{o \mapsto e\}] & \Rightarrow B[store \backslash store \oplus \{o \mapsto f\}] \\
  \equiv & \quad \text{Object-Refinement Monotonic Predicate (7.8)} \\
  \quad \text{Pragmatic constraint: all predicates, including } B, \text{ are object-refinement monotonic.} \\
  \quad \quad (store \oplus \{o \mapsto e\}) & \sqsubseteq (store \oplus \{o \mapsto f\}) \\
  \equiv & \quad \text{Object-Refinement Algorithmic (7.1)} \\
  e & \sqsubseteq^e f
\end{align*}
\]

QED

Proof of 7.25 from p90 (Object-Refinement Specification (Semantics for References))

Duplicate (Object-Refinement Specification (Semantics for References)) of 7.25 on page 90.
This refinement is analogous to Simple Specification (B.52) except this rule is applicable to a semantics for references. Assume a reference variable \( o \) and an expression \( e \) which may not include initial variables. The alteration of the store at \( o \) to establish \( o \uparrow \) as an object-refinement of \( e \) can be implemented by assigning \( e \) to \( o \uparrow \):

\[
o \uparrow :: [e_0 \sqsubseteq^e o \uparrow] \sqsubseteq o \uparrow := e
\]

where \( e_0 \) is \( e \) with all dereferences of \( store \) replaced with corresponding dereferences of \( store_0 \).
For any object-refinement monotonic predicate $p$, 

\[
o^0 \models [e_0 \sqsubseteq^* o^1] \quad p
\]

$\equiv$ Modified Object Specification (Semantics for References) (7.15)

\[
\begin{align*}
\text{store:} & \quad \left[ e_0 \sqsubseteq^* o^1 \land \\
& \quad \forall n \in \text{dom(store)} \setminus \text{dom}(\zeta(\{o^1\}, \emptyset, \emptyset)) \bullet \text{store}(n) \sqsubseteq^* \text{store}_0(n) \\
& \quad \forall \text{refs} \in \text{dom}(\zeta(\{o\}, \emptyset, \emptyset)) \bullet \\
& \quad \forall a \in \text{attributes}((\text{refs})\setminus(\zeta(\{o\}, \emptyset, \emptyset)))(\text{refs}) \bullet \\
& \quad \text{refs}^\uparrow . a \sqsubseteq^* \text{refs}^\uparrow_0 . a \\
\end{align*}
\]

$\equiv$ As $\zeta(\{o\}, \emptyset, \emptyset) = \{o \mapsto \text{attributes}(o)\}$

\[
\begin{align*}
\text{store:} & \quad \left[ e_0 \sqsubseteq^* o^1 \land \\
& \quad \forall n \in \text{dom(store)} \setminus \{o\} \bullet \text{store}(n) \sqsubseteq^* \text{store}_0(n) \\
& \quad \forall a \in \text{attributes}(o) \bullet \\
& \quad o^1 . a \sqsubseteq^* o^1_0 . a \\
\end{align*}
\]

$\equiv$ Singleton set

\[
\begin{align*}
\text{store:} & \quad \left[ e_0 \sqsubseteq^* o^1 \land \\
& \quad \forall n \in \text{dom} \text{store} \setminus \{o\} \bullet \text{store}(n) \sqsubseteq^* \text{store}_0(n) \\
& \quad \forall a \in \text{attributes}(a) \bullet \\
& \quad o^1 . a \sqsubseteq^* o^1_0 . a \\
\end{align*}
\]

$\equiv$ Vacuous set

\[
\begin{align*}
\text{store:} & \quad \left[ e_0 \sqsubseteq^* o^1 \land \\
& \quad \forall n \in \text{dom} \text{store} \setminus \{o\} \bullet \text{store}(n) \sqsubseteq^* \text{store}_0(n) \\
\end{align*}
\]

$\equiv$ Specification Statement (B.47)

\[
\begin{align*}
& \quad (\forall \text{store} \bullet (e_0 \sqsubseteq^* o^1 \land \forall n \in \text{dom} \text{store} \setminus \{o\} \bullet \text{store}(n) \sqsubseteq^* \text{store}_0(n)) \\
& \Rightarrow p)[\text{store}_0 \setminus \text{store}]
\end{align*}
\]

$\Rightarrow$ Universal Quantification Elimination (B.15) using $\text{store} \oplus \{o \mapsto e\}$

\[
\begin{align*}
& \quad ((e_0 \sqsubseteq^* o^1 \land \forall n \in \text{dom} \text{store} \setminus \{o\} \bullet \text{store}(n) \sqsubseteq^* \text{store}_0(n)) \\
& \Rightarrow p)[\text{store}_0 \setminus \text{store} \oplus \{o \mapsto e\}][\text{store}_0 \setminus \text{store}]
\end{align*}
\]

$\equiv$ Object Dereference (6.13)

\[
\begin{align*}
& \quad ((e_0 \sqsubseteq^* \text{store}(o) \land \\
& \quad \forall n \in \text{dom} \text{store} \setminus \{o\} \bullet \text{store}(n) \sqsubseteq^* \text{store}_0(n)) \\
& \Rightarrow p)[\text{store}_0 \setminus \text{store} \oplus \{o \mapsto e\}][\text{store}_0 \setminus \text{store}]
\end{align*}
\]

$\equiv$ Substitutions

\[
\begin{align*}
& \quad (e_0)[\text{store}_0 \setminus \text{store}] \sqsubseteq^* (\text{store} \oplus \{o \mapsto e[\text{store}_0 \setminus \text{store}]\}(o)) \land \\
& \quad \forall n \in \text{dom} \text{store} \setminus \{o \mapsto e[\text{store}_0 \setminus \text{store}]\} \bullet \\
& \quad (\text{store} \oplus \{o \mapsto e[\text{store}_0 \setminus \text{store}]\}(n) \sqsubseteq^* \text{store}(n)) \\
& \Rightarrow p)[\text{store}_0 \setminus \text{store} \oplus \{o \mapsto e[\text{store}_0 \setminus \text{store}]\}]
\end{align*}
\]

$\equiv$ $\text{store}_0$ does not occur in $e$.

\[
\begin{align*}
& \quad (e \sqsubseteq^* e \land \forall n \in \text{dom} \text{store} \setminus \{o\} \bullet \text{store}(n) \sqsubseteq^* \text{store}(n)) \\
& \Rightarrow p)[\text{store}_0 \setminus \text{store} \oplus \{o \mapsto e\}]
\end{align*}
\]

$\equiv$ Object-Refinement Reflexive (B.64)

\[
\begin{align*}
& \quad p)[\text{store}_0 \setminus \text{store} \oplus \{o \mapsto e\}]
\end{align*}
\]

$\equiv$ Assignment (B.30)

\[
\begin{align*}
& \quad (\text{store} := \text{store} \oplus \{o \mapsto e\} \ p)
\end{align*}
\]

$\equiv$ Accessed Function Assignment (6.12), Object Dereference (6.13)

\[
\begin{align*}
& \quad (o^1 := e) \ p
\end{align*}
\]
The proof is concluded by Definition Refinement (4.32).

**QED**

**Proof of 7.29 from p92 (Introduce Field Update (Semantics for Values))**

**Duplicate (Introduce Field Update (Semantics for Values)) of 7.29 on page 92.**

A field update can be introduced to force a field \( fld \) of object \( o \) to be an object-refinement of an expression \( e \) that may not include initial variables.

\[
o.\ fld:: [e[o\neg o_0] \sqsubseteq o.\ fld] \sqsubseteq o.\ fld := e
\]

\[
o.\ fld:: [e[o\neg o_0] \sqsubseteq o.\ fld]
\equiv \text{Modified Object Specification (Semantics for Values) (7.14)}
\equiv o:\ [e[o\neg o_0] \sqsubseteq o.\ fld \land (\forall a \in \poc\{\{o\}\}\{o, o.\ fld\} \bullet a \sqsubseteq a_0)]
\equiv \text{Strengthen Postcondition (B.54)}
\equiv o:\ [(o_0.o.\ fld \equiv e[o\neg o_0]) \sqsubseteq o]
\equiv \text{Object-Refinement Specification (7.13)}
\equiv o:: (o_0.o.\ fld \equiv e)
\equiv \text{Object Field Update (Semantics for Values) (6.8)}
\equiv o.\ fld := e
\]

**QED**

**Proof of 7.30 from p92 (Introduce Field Update (Semantics for References))**

**Duplicate (Introduce Field Update (Semantics for References)) of 7.30 on page 92.**

For a semantics for references, a field update can be introduced to force a field \( fld \) of object \( o \) to be an object-refinement of an expression \( e \) which may not include initial variables:

\[
o_0.\ fld:: [e_0 \sqsubseteq o_0.\ fld] \sqsubseteq o_0.\ fld := e
\]

where \( e_0 \) is \( e \) with dereferences of \( store \) replaced by corresponding dereferences of \( store_0 \).

The proof is almost identical to that of Introduce Field Update (Semantics for References) A (6.19). The differences are that Theorem Modified Object Specification (Semantics for References) (7.15) is used instead of Theorem Object Specification (Semantics for References) (6.16), the object-refinement relation is used instead of equality, and Theorem Object-Refinement Specification (Semantics for References) (7.25) is used instead of Theorem Simple Specification (B.52).

**QED**
APPENDIX D. PROOFS

Proof of 7.31 from p93 (Object-Refine Field Update (Semantics for Values))

Duplicate (Object-Refine Field Update (Semantics for Values)) of 7.31 on page 93.

This refinement rule allows the expression being assigned in a field update to be object-refined.

\[
\begin{align*}
e & \sqsubseteq^e f \\
(o.fld := e) & \sqsubseteq (o.fld := f)
\end{align*}
\]

Proof of 7.32 from p93 (Object-Refine Field Update (Semantics for References))

Duplicate (Object-Refine Field Update (Semantics for References)) of 7.32 on page 93.

This refinement rule allows the expression being assigned in a field update to be object-refined for a semantics for references.

\[
\begin{align*}
e & \sqsubseteq^e f \\
(o^\uparrow.fld := e) & \sqsubseteq (o^\uparrow.fld := f)
\end{align*}
\]

Proof of 7.33 from p93 (Introduce Method Call (Semantics for Values))
Duplicate (Introduce Method Call (Semantics for Values)) of 7.33 on page 93.

For a semantics for values, given object $o$:

$$(o \odot m)[self \setminus o] \sqsubseteq o.\text{call } m$$

$$(o \odot m)[self \setminus o]$$

\[ \text{Refinement Calculus: Effectively skip}\]

Enter (B.33), Exit (B.35)

For fresh $v$ and $self$ is not free in $(o \odot m)[self \setminus o]$

$\langle [\textbf{var } v : Self := self;\]
\textbf{exit } self; (o \odot m)[self \setminus o]; \textbf{enter } self : Self := v\rangle$ \[ \text{Refinement Calculus: } self \text{ is reestablished.} \]

$self$ is not free in $(o \odot m)[self \setminus o]$

Assignment (B.30)

$\langle [\textbf{var } v : Self := self;\]
\textbf{exit } self;\]
$\langle [\textbf{var } self : O \bullet\]
\textbf{self := o; } (self \odot m)[self \setminus o]\rangle$

$\langle \textbf{enter } self : Self := v\rangle$ \[ \text{Refinement Calculus: } self \text{ reestablished.} \]

Assignment (B.30)

$\langle [\textbf{var } v : Self := self;\]
\textbf{exit } self;\]
$\langle [\textbf{var } self : O \bullet\]
\textbf{self := o; } self \odot m; o := self\rangle$

$\langle \textbf{enter } self : Self := v\rangle$ \[ \text{Assignement (B.30)} \]

$\equiv \text{Internal Object Method Invocation (Semantics for Values) (6.10)}$

$\langle [\textbf{var } v : Self := self;\]
\textbf{exit } self;\]
$\langle [\textbf{var } self : O \bullet\]
\textbf{self := o; } \textbf{call } m; o := self\rangle$

$\langle \textbf{enter } self : Self := v;\rangle$ \[ \equiv \text{External Object Method Invocation (Semantics for Values) (6.11)}\]

\[o.\text{call } m\]

\[QED\]

Proof of 7.34 from p93 (Introduce Method Call (Semantics for References))
Duplicate (Introduce Method Call (Semantics for References)) of 7.34 on page 93.

Similarly, for a semantics for references, given an object $o^\uparrow$:

$$(o^\uparrow \odot m)[\text{self} \setminus o] \subseteq o^\uparrow . \text{call } m$$

\[
(o^\uparrow \odot m)[\text{self} \setminus o] \\
\subseteq \text{Refinement Calculus: effectively skip.} \\
\text{For fresh } v \text{ and self not free in } (o^\uparrow \odot m)[\text{self} \setminus o] \\
\begin{array}{l}
\left[ \text{var } v : \text{Ref} := \text{self}; \\
\quad (o^\uparrow \odot m)[\text{self} \setminus o]; \text{ self} := v; \\
\right]
\end{array} \\
\subseteq \text{Refinement Calculus: self is established.} \\
\begin{array}{l}
\left[ \text{var } v : \text{Ref} := \text{self}; \\
\quad \text{self} := o; \ (o^\uparrow \odot m)[\text{self} \setminus o]; \text{ self} := v; \\
\right]
\end{array} \\
\equiv \text{self and } o \text{ are aliased.} \\
\begin{array}{l}
\left[ \text{var } v : \text{Ref} := \text{self}; \\
\quad \text{self} := o; \ (o^\uparrow \odot m); \text{ self} := v; \\
\right]
\end{array} \\
\equiv \text{Internal Method Invocation (Semantics for References) (6.20)} \\
\begin{array}{l}
\left[ \text{var } v : \text{Ref} := \text{self}; \\
\quad \text{self} := o; \ (o^\uparrow \odot m); \text{ self} := v; \\
\right]
\end{array} \\
\equiv \text{External Method Invocation (Semantics for References) (6.21)} \\
\begin{array}{l}
\text{ o}^\uparrow . \text{call } m
\end{array}
\]

QED

Proof of 7.35 from p93 (Value Parameterised Method Call)
Duplicate (Value Parameterised Method Call) of 7.35 on page 93.

This rule is essentially that of Morgan’s Law 11.2 [Mor94]. The rule permits the use of a value parameter in the method being introduced. Assume an actual value parameter \( a \), a formal value parameter \( f \) (both disjoint from \( w \)), and a method \( m \) of object \( o \):

\[
\text{method } m(\text{value } f : F)
\]

For a semantics for values, if

\[
w. f : [\text{pre}, \text{post}] \subseteq (o \circ m)[\text{self} \setminus o]
\]

with \( \text{post} \) containing no \( f \), then

\[
w. a : [\text{pre}[f \setminus a], \text{post}[f_0 \setminus a_0]] \subseteq o \cdot \text{call } m(a)
\]

Similarly, for a semantics for references, if

\[
w. f : [\text{pre}, \text{post}] \subseteq (o^\uparrow \circ m)[\text{self} \setminus o]
\]

with \( \text{post} \) containing no \( f \), then

\[
w. a : [\text{pre}[f \setminus a], \text{post}[f_0 \setminus a_0]] \subseteq o^\uparrow \cdot \text{call } m(a)
\]

The proof uses Theorems 7.33, 7.34 and Morgan’s [Mor94] Law 11.2.

\[ QED \]

Proof of B.62 from p187 (Result Parameterised Method Call)
Duplicate (Result Parameterised Method Call) of B.62 on page 187.
This rule permits the use of a result parameter in the method being introduced. Assume a formal result parameter $f$, an actual result parameter $a$ (both disjoint from $w$) where $a$ is neither $self$ or $o$, and a method $m$ of object $o$. For a semantics for values, if

$$w.f: [pre, post[a\setminus f]] \subseteq (o \circ m)[self \setminus o]$$

with $pre$ containing no $f$ and neither $f$ nor $f_0$ occurring in $post$, then

$$w, a: [pre, post] \subseteq o.call m(a)$$

Similarly, for a semantics for references, if

$$w.f: [pre, post[a\setminus f]] \subseteq (o \lhd m)[self \setminus o]$$

with $pre$ containing no $f$ and neither $f$ nor $f_0$ occurring in $post$, then

$$w, a: [pre, post] \subseteq o.lcall m(a)$$

The nomenclature result is used to denote a result parameter.

The proof uses Theorems 7.33, 7.34 and Morgan’s [Mor94] Law 11.4.

QED

D.8 Data refinement for Objects Proofs

Theorem D.5 (Merge Specification and Assignment) Given distinct variables $w, t$ and $o$ with $o_0$ not free in $post$:

$$w: \begin{array}{c} \frac{pre}{post} \end{array} \Rightarrow o := t \equiv \begin{array}{c} o \vdash \frac{pre}{post\setminus o_0} \land o = t \end{array}$$

Proof
\[ w : [\text{pre}, \text{post}] : o \coloneqq tp \]
\[ \equiv \text{Assignment (B.30)} \]
\[ \text{Specification Statement (B.47)} \]
\[ \text{pre} \land (\forall w \bullet \text{post} \Rightarrow p[o\setminus t])[w_0\setminus w] \]
\[ \equiv \text{Universal Quantification One Point Rule (B.7)} \]
\[ \text{pre} \land (\forall w \bullet \text{post} \Rightarrow (\forall o \bullet o = t \Rightarrow p))[w_0\setminus w] \]
\[ \equiv o_0 \text{ is not free in post or } p \]
\[ \text{pre} \land (\forall w \bullet \text{post}[o\setminus o_0] \Rightarrow (\forall o \bullet o = t \Rightarrow p))[w_0, o_0\setminus w, o] \]
\[ \equiv \text{Partially Superfluous Universal Quantification Implication (B.12)} \]
\[ \text{pre} \land (\forall w, o \bullet \text{post}[o\setminus o_0] \Rightarrow (o = t \Rightarrow p))[w_0, o_0\setminus w, o] \]
\[ \equiv \text{Double Implication (B.5)} \]
\[ \text{pre} \land (\forall w, o \bullet \text{post}[o\setminus o_0] \land o = t \Rightarrow p)[w_0, o_0\setminus w, o] \]
\[ \equiv \text{Specification Statement (B.47)} \]
\[ w, o : [\text{pre}, \text{post}[o\setminus o_0] \land o = t] p \]

\[ QED \]

**Proof of 7.42 from p112 (Data Refine New)**

Duplicate (Data Refine New) of 7.42 on page 112.

Given \( \text{spec} \preceq_{\text{impl}} \text{impl} \) and

\[ \text{REP } p \equiv (\exists \text{store}_s \bullet o\vdash_{\text{store}_s} \preceq_{\text{impl}} \circ\vdash_{\text{store}_s, \{o\} \lhd \text{store}_s} = (\{o\} \lhd \text{store}_i) \land p ) \]

then

\[ o : \text{new spec} \preceq_{\text{REP}} o : \text{new impl} \]

\[ o : \text{new}_{\text{store}_s}, \text{spec} \]
\[ \equiv \text{New Operator (7.26)} \]

\[ \left[ \text{var } t : \text{Ref} \bullet \right] \]
\[ \left[ (\forall j \bullet t \not\in \text{dom } \text{store}_j) ; t\vdash_{\text{store}_s} := \text{spec}; o := t \right] \]

Using Data refine Sequential Composition (2.5) this can be broken into the data refinement of each statement. The first data refines, using Data Refine Specification (2.4) to

\[ t : [(\forall j \bullet t \not\in \text{dom } \text{store}_j)] \]
The second and third statements data refine as follows:

\[ t \uparrow_{\text{store}} := \text{spec}; \quad o := t \]
\[ \equiv \text{Accessed Function Assignment (6.12)} \]
\[ \text{store} \downarrow := \text{store} \oplus \{ t \mapsto \text{spec}\}; \quad o := t \]
\[ \equiv \text{Object-Refinement Specification (7.13)} \]
\[ \text{store}_s : [ \text{store} \cong^s \text{store}_{s_0} \oplus \{ t \mapsto \text{spec}[\text{store}, o \\setminus \text{store}_{s_0}, o_0]\} ]; \quad o := t \]
\[ \equiv \text{Merge Specification and Assignment (D.5)} \]
\[ \text{store}_s, o : [ \text{store} \cong^s \text{store}_{s_0} \oplus \{ t \mapsto \text{spec}[\text{store}, o \\setminus \text{store}_{s_0}, o_0]\} \land o = t ] \]
\[ \equiv \text{Property of function override} \]
\[ \text{store}_s, o : \left[ \forall k \in \text{dom}(\text{store}_{s_0}) \{ o \} \bullet k \uparrow_{\text{store}} \equiv^s k \uparrow_{\text{store}_0} \land \right. \]
\[ o \uparrow_{\text{store}} \equiv^s \text{spec}[\text{store}, o \\setminus \text{STORES}, o_0] \land o = t \]
\[ \leq_{\text{REP}} \text{Data Refine Specification (2.4)} \]
\[ [[ \text{con STORES, store}_s \bullet \right. \]
\[ o \uparrow_{\text{store}} \leq_{\text{ai}}^s o \uparrow_{\text{store}} \land \]
\[ \{ o \} \leftarrow \text{store}_s = \{ o \} \leftarrow \text{store}_i \land \right. \]
\[ \text{STORES} = \text{store}_s \]
\[ ] \]
\[ \equiv \text{Using assumption: } \text{spec} \leq_{\text{ai}}^s \text{impl} \]
\[ [[ \text{con STORES, store}_s \bullet \right. \]
\[ \text{spec} \leq_{\text{ai}}^s \text{impl} \land o \uparrow_{\text{store}} \leq_{\text{ai}}^s o \uparrow_{\text{store}} \land \]
\[ \{ o \} \leftarrow \text{store}_s = \{ o \} \leftarrow \text{store}_i \land \right. \]
\[ \text{STORES} = \text{store}_s \]
\[ ] \]
\[ \equiv \text{Strengthen Postcondition (B.54) with } o_0 \uparrow_{\text{store}} \equiv^s o_0 \uparrow_{\text{store}_0} \text{ and from the precondition: } \{ o_0 \} \leftarrow \text{STORES} = \{ o_0 \} \leftarrow \text{store}_i \]
\[ \text{Weaken Precondition (B.53)} \]
\[ \text{Remove Logical Constant (B.45)} \]
\[ [[ \text{con STORES} \bullet \right. \]
\[ \text{spec}[\text{store} \setminus \text{STORES}] \leq_{\text{ai}}^s \text{impl} \]
\[ \exists \text{store}_s \bullet o \uparrow_{\text{store}} \leq_{\text{ai}}^s o \uparrow_{\text{store}} \land \]
\[ \{ o \} \leftarrow \text{store}_s = \{ o \} \leftarrow \text{store}_i \land \right. \]
\[ \forall k \in \text{dom}(\text{store}_i) \{ o \} \bullet k \uparrow_{\text{store}_i} \equiv^s k \uparrow_{\text{store}_0} \land \]
\[ o \uparrow_{\text{store}_i} \equiv^s \text{spec}[\text{store}_s, o \\setminus \text{STORES}, o_0] \land o = t \]
\[ ] \]
APPENDIX D. PROOFS

\[
\begin{align*}
\Rightarrow & \text{ Strengthen Postcondition (B.54) with } \\
\text{store}_i &= \text{store}_i \oplus \{ o \mapsto \text{spec}[\text{store}_i, o \setminus \text{STORES}, o_0] \} \\
\text{Object Dereference (6.13)}
\end{align*}
\]

\[
\begin{align*}
\text{[[ con STORES •}} \\
\exists \text{store}_i \cdot \text{store}_i &= \text{store}_i \oplus \{ o \mapsto \text{spec}[\text{store}_i, o \setminus \text{STORES}, o_0] \} \\
\text{spec}[\text{store}_i, o \setminus \text{STORES}, o_0] &\preceq_o o \uparrow_{\text{store}_i} \land \\
\{ o \} &\ll \text{store}_i = \{ o \} \ll \text{store}_i \land \\
\forall k \in \text{dom}(\text{store}_i) \setminus \{ o \} \cdot k \uparrow_{\text{store}_i} \exists s k \uparrow_{\text{store}_i} o \land \\
\text{spec}[\text{store}_i, o \setminus \text{STORES}, o_0] &\equiv^* \text{spec}[\text{store}_i, o \setminus \text{STORES}, o_0] \land \\
o &\equiv t
\end{align*}
\]

\[
\text{[[ con STORES •}}
\]

\[
\begin{align*}
\Rightarrow & \text{ Strengthen Postcondition (B.54) with } \text{impl}[\text{store}_i, o \setminus \text{store}_i, o_0] \subset^* o \uparrow_{\text{store}_i} \\
\text{and using the preconidion} \\
\text{spec}[\text{store}_i, o \setminus \text{STORES}, o_0] &\preceq_o \text{impl}[\text{store}_i, o \setminus \text{store}_i, o_0] \text{ as follows:} \\
\text{spec}[\text{store}_i, o \setminus \text{STORES}, o_0] &\preceq_o \text{impl}[\text{store}_i, o \setminus \text{store}_i, o_0] \land \\
\text{impl}[\text{store}_i, o \setminus \text{store}_i, o_0] &\subset^* o \uparrow_{\text{store}_i} \\
\Rightarrow & \text{ Object-Refinement Monotonic Predicate (7.8)} \\
\text{spec}[\text{store}_i, o \setminus \text{STORES}, o_0] &\preceq_o o \uparrow_{\text{store}_i}
\end{align*}
\]

\[
\begin{align*}
\text{[[ con STORES •}} \\
\forall k \in \text{dom}(\text{store}_i) \setminus \{ o \} \cdot k \uparrow_{\text{store}_i} \exists s k \uparrow_{\text{store}_i} o \land \\
o &\equiv t
\end{align*}
\]

\[
\Rightarrow & \text{ Weaken Precondition (B.53)} \\
& \text{Remove Logical Constant (B.45)}
\]

\[
\begin{align*}
\text{store}_i, o: \left[ \text{impl}[\text{store}_i, o \setminus \text{store}_i, o_0] \subset^* o \uparrow_{\text{store}_i} \land \\
\forall k \in \text{dom}(\text{store}_i) \setminus \{ o \} \cdot k \uparrow_{\text{store}_i} \exists s k \uparrow_{\text{store}_i} o \land \\
o &\equiv t \right]
\end{align*}
\]

\[
\Rightarrow & \text{ Property of function override.} \\
\text{store}_i, o: \left[ \text{store}_i \not\subset^* \text{store}_i \oplus \{ o \mapsto \text{impl}[\text{store}_i, o \setminus \text{store}_i, o_0] \} \land o = t \right]
\]

\[
\Rightarrow & \text{ Introduce Sequential Composition (B.40)} \\
& \text{Contract Frame (B.51)} \\
& \text{Strengthen Postcondition (B.54)} \\
& \text{Weaken Precondition (B.53)}
\]

\[
\begin{align*}
\text{store}_i: \left[ \text{store}_i \not\subset^* \text{store}_i \oplus \{ t \mapsto \text{impl}[\text{store}_i, o \setminus \text{store}_i, o_0] \} \right] \\
o: \left[ o = t \right]
\end{align*}
\]
APPENDIX D. PROOFS

≡ Object-Refinement Specification (7.13)
Accessed Function Assignment (6.12)
Simple Specification (B.52)

\[ t |_{store_i} := \text{impl}; \ o := t \]

\[ \text{QED} \]

Proof of 7.43 from p113 (Data Refine Store in Method Call)

Duplicate (Data Refine Store in Method Call) of 7.43 on page 113.
Given public method \( m \), and

\[ REP \ p \equiv (\exists \ store_s \bullet o |_{store_s} \leq_{ai} o |_{store_i} \wedge (\{o\} \sqsubseteq store_s) = (\{o\} \sqsubseteq store_i) \wedge p ) \]

then

\[ o |_{store_i}, \text{call} \ m \leq_{\text{REP}} o |_{store_i}, \text{call} \ m \]

Using External Method Invocation (Semantics for References) (6.21), Internal Method Invocation (Semantics for References) (6.20), Refinement (4.32), and Data Refinement (2.3), the proof reduces to showing

\[ (REP; \ self |_{store, \circ} m) \ p \Rightarrow (self |_{store, \circ} m; \ REP) \ p \]

in a context where \( self = o \) can be assumed.

\[ (REP; \ self |_{store, \circ} m) \ p \equiv \text{Definition of REP} \]

\[ (\exists \ store_s \bullet \]

\[ o |_{store_s} \leq_{ai} o |_{store_i} \wedge (\{o\} \sqsubseteq store_s = (\{o\} \sqsubseteq store_i) \wedge (self |_{store, \circ} m)(p)) \]

From \( o |_{store_s} \leq_{ai} o |_{store_i} \wedge o |_{store, \circ} m \leq_{\text{odrepr} p} o |_{store, \circ} m \) is deduced where \( \text{odrep} \) (and \( \text{rep}'' \)) are defined in Object-Data-Refinement (7.39).

\[ \text{rep}'' q \equiv (\exists \ store_s \bullet ai \wedge (\{self\} \sqsubseteq store_s = (\{self\} \sqsubseteq store_i) \wedge q) \]

\[ \text{odrepr} p \equiv (\exists \ store_s \bullet o |_{store_s} \leq_{\text{rep}''} o |_{store_i} \wedge (\{self\} \sqsubseteq store_s = (\{self\} \sqsubseteq store_i) \wedge p) \]

\( REP \ p \) reduces to the following using Object-Data-Refinement (7.39).

\[ REP \ p \equiv (\exists \ store_s \bullet o |_{store_s} \leq_{\text{odrepr} p} o |_{store_i} \wedge (\{o\} \sqsubseteq store_s = (\{o\} \sqsubseteq store_i) \wedge p) \]

There are no language constructs that modify an object’s methods. Also, for an object-data-refinement to be shown, all public methods must be data refined under \( \text{odrepr} \). Together with the assumption \( self = o \), these properties are used to generalise

\[ o |_{store, \circ} m \leq_{\text{odrepr} p} o |_{store, \circ} m \]
to

\[ o_{\text{store}} \downarrow_m \leq_{\text{REP}} o_{\text{store}} \downarrow_m \]

Consequently, the proof proceeds as follows.

\[ \Rightarrow \text{Data Refinement (2.3)} \]

\[ (\text{store}(o) \downarrow_m) (\exists \text{ store}_i \bullet \]

\[ o_{\text{store}_i} \leq_{\text{ai}} o_{\text{store}_i} \wedge \{o\} \triangleq \text{ store}_s = \{o\} \triangleq \text{ store}_i \wedge p \]

\[ \equiv \text{Definition of REP} \]

\[ (\text{store}(o) \downarrow_m; \text{ REP}) p \]

\[ \text{QED} \]

**Proof of 7.44 from p113 (Data Refine Object Assignment)**

Duplicate (Data Refine Object Assignment) of 7.44 on page 113.

Given \( e \leq_{\text{ai}} f \), and

\[ \text{REP } p \equiv (\exists \text{ store}_i \bullet o_{\text{store}_i} \leq_{\text{ai}} o_{\text{store}_i} \wedge \{o\} \triangleq \text{ store}_s = \{o\} \triangleq \text{ store}_i \wedge p ) \]

then

\[ o_{\text{store}_i} := e \leq_{\text{REP}} o_{\text{store}_i} := f \]

\[ o_{\text{store}_i} := e \]

\[ \equiv \text{Accessed Function Assignment (6.12)} \]

\[ \text{store}_i := \text{store}_i \oplus \{o \mapsto e\} \]

\[ \equiv \text{Object-Refinement Specification (7.13)} \]

\[ \text{store}_i; \left[ \text{store}_i \geq_{\text{ai}} \text{ store}_s \oplus \{o \mapsto e[\text{store}_s \backslash \text{store}_i] \} \right] \]

\[ \leq_{\text{REP}} \text{ Data Refine Specification (2.4)} \]

\[ \left[ [ \text{ con store}_s, \text{ STORE}_s \bullet \right] \]

\[ o_{\text{store}_s} \leq_{\text{ai}} o_{\text{store}_s} \wedge \{o\} \trianglerighteq \text{ store}_s = \{o\} \trianglerighteq \text{ store}_i \wedge \]

\[ \text{STORE}_s = \text{ store}_s \]

\[ \exists \text{ store}_s \bullet \]

\[ o_{\text{store}_s} \leq_{\text{ai}} o_{\text{store}_s} \wedge \]

\[ \{o\} \triangleq \text{ store}_s = \{o\} \triangleq \text{ store}_i \wedge \]

\[ (\text{store}_s \geq_{\text{ai}} \text{ store}_s \oplus \{o \mapsto e[\text{store}_s \backslash \text{store}_i] \}) [\text{store}_s \backslash \text{STORE}_s] \]
\[\equiv\text{ Substitution} \]
\[\frac{\text{con } store_s, \text{STORE}_s \cdot}{\exists store_s \cdot \begin{array}{l}
o^{\downarrow}_{\text{store}}, \preceq_{\text{ui}} o^{\downarrow}_{\text{store}}, \land \{o\} \ll store_s = \{o\} \ll store_i \land 
\text{STORE}_s = store_s 
\end{array}}{\{o\} \ll store_s, \exists^* \{o\} \ll store_s \land \text{STORE}_s \supseteq store_s \cup \{o \mapsto e[\text{store}_s, \text{STORE}_s]\}}\]

\[\equiv\text{ Strengthen Postcondition (B.54)} \]
\[\frac{\text{con } store_s, \text{STORE}_s \cdot}{\exists store_s \cdot \begin{array}{l}
o^{\downarrow}_{\text{store}}, \preceq_{\text{ui}} o^{\downarrow}_{\text{store}}, \land \{o\} \ll store_s = \{o\} \ll store_i \land 
\text{STORE}_s = store_s 
\end{array}}{\{o\} \ll store_s, \exists^* \{o\} \ll store_s \land \text{STORE}_s \supseteq store_s \cup \{o \mapsto e[\text{store}_s, \text{STORE}_s]\}}\]

\[\equiv\text{ Existential Quantification One Point Rule (B.6)} \]
\[\frac{\text{con } store_s, \text{STORE}_s \cdot}{\begin{array}{l}
o^{\downarrow}_{\text{store}}, \preceq_{\text{ui}} o^{\downarrow}_{\text{store}}, \land \{o\} \ll store_s = \{o\} \ll store_i \land 
\text{STORE}_s = store_s 
\end{array}}{\begin{array}{l}
e[\text{store}_s, \text{STORE}_s]^{\downarrow}_{\text{store}}, \preceq_{\text{ui}} o^{\downarrow}_{\text{store}}, \land 
\{o\} \ll store_s = \{o\} \ll store_i \land 
\text{STORE}_s \cup \{o \mapsto e[\text{store}_s, \text{STORE}_s]\} \exists^* 
\end{array}}\]

\[\equiv\text{ Strengthen Postcondition (B.54) with precondition STORE}_s = store_s \]
\[\begin{array}{l}
\text{Weaken Precondition (B.53)} \\
\text{Remove Logical Constant (B.45)} \\
\end{array}\]

\[\frac{\text{con } store_s \cdot}{\begin{array}{l}
o^{\downarrow}_{\text{store}}, \preceq_{\text{ui}} o^{\downarrow}_{\text{store}}, \land \{o\} \ll store_s = \{o\} \ll store_i 
\end{array}}{\begin{array}{l}
e[\text{store}_s, \text{STORE}_s]^{\downarrow}_{\text{store}}, \preceq_{\text{ui}} o^{\downarrow}_{\text{store}}, \land 
\{o\} \ll store_s = \{o\} \ll store_i 
\end{array}}\]

\[\equiv\text{ Object-Refinement Algorithmic (7.1)} \]
\[\frac{\text{con } store_s \cdot}{\begin{array}{l}
o^{\downarrow}_{\text{store}}, \preceq_{\text{ui}} o^{\downarrow}_{\text{store}}, \land 
\end{array}}{\begin{array}{l}
e[\text{store}_s, \text{STORE}_s]^{\downarrow}_{\text{store}}, \preceq_{\text{ui}} o^{\downarrow}_{\text{store}}, \land 
\{o\} \ll store_i \exists^* \{o\} \ll store_s \land e \exists^* e 
\end{array}}\]
\[ \square \text{Strengthen Postcondition (B.54) with precondition}
\]
\[ \text{Weaken Precondition (B.53)} \]
\[ \text{Introduce assumption } e \preceq_{\text{ai}} f \text{ into precondition} \]
\[ \begin{align*}
\left[ \text{con } \mathit{store}, \bullet \right. \\
\left. \begin{array}{c}
\mathit{store}; \left[ e \preceq_{\text{ai}} f \right] \\
\left[ e \upharpoonright_{\mathit{store},} \preceq_{\text{ai}} o \upharpoonright_{\mathit{store},} \land \{ o \} \subset \mathit{store}; \exists^* \{ o \} \subset \mathit{store}_{i 0} \right] \end{array} \right]
\end{align*} \]
\[ \square \text{Remove Logical Constant (B.45)} \]
\[ \text{Strengthen Postcondition (B.54) with } \mathit{store}_i(o) \exists^* f[\mathit{store}_i \backslash \mathit{store}_{i 0}] \]
\[ \text{and from the precondition } e \preceq_{\text{repr}} f[\mathit{store}_i \backslash \mathit{store}_{i 0}] : \]
\[ \mathit{store}_i(o) \exists^* f[\mathit{store}_i \backslash \mathit{store}_{i 0}] \land e \preceq_{\text{repr}} f[\mathit{store}_i \backslash \mathit{store}_{i 0}] \]
\[ \Rightarrow \text{Object-Refinement Monotonic Predicate (7.8)} \]
\[ \mathit{store}_i(o) \exists^* f[\mathit{store}_i \backslash \mathit{store}_{i 0}] \land e \preceq_{\text{repr}} \mathit{store}_i(o) \]
\[ \mathit{store}_i; \left[ \mathit{store}_i(o) \exists^* f[\mathit{store}_i \backslash \mathit{store}_{i 0}] \land \{ o \} \subset \mathit{store}_i \exists^* \{ o \} \subset \mathit{store}_{i 0} \right] \]
\[ \square \text{Object-Refinement Algorithmic (7.1)} \]
\[ \mathit{store}_i; \left[ \mathit{store}_i \exists^* \mathit{store}_{i 0} \oplus \{ o \mapsto f[\mathit{store}_i \backslash \mathit{store}_{i 0}] \} \right] \]
\[ \square \text{Object-Refinement Specification (7.13)} \]
\[ \mathit{store}_i \text{ is not free in } f[\mathit{store}_i \backslash \mathit{store}_{i 0}] \]
\[ \mathit{store}_i := \mathit{store}_i \oplus \{ o \mapsto f \} \]

\textit{QED}

\textbf{Proof of 7.46 from p113 (Data Refine Object Specifications (Semantics for Values))}
Duplicate (Data Refine Object Specifications (Semantics for Values)) of 7.46 on page 113.

Assume an abstract object \( spec \), a concrete object \( impl \), abstraction invariant \( AI \), abstract fields in the frame \( af \), abstract fields not in the frame \( anf \), introduced concrete fields in the frame \( bf \), concrete fields not in the frame \( bnf \), common fields in the frame \( gf \), and common fields not in the frame \( gnf \). Given \( ai \equiv AI \land spec.gf \sqsubseteq impl.gf \land spec.gnf \sqsubseteq impl.gnf \), and \( REP\ p \equiv (\exists\ spec \bullet ai \land p) \), an object specification data refines as follows:

\[
\begin{align*}
\text{spec} \cdot\ af,\ spec.gf :: [pre,\ post] \\
\overset{\text{REP}}{\Rightarrow} \left[ \begin{array}{c}
\text{con spec}_0 \\
\text{impl} \cdot\ bf,\ impl.gf :: \\
\end{array} \right]
\end{align*}
\]

provided

\[
\text{ai}_0 \Rightarrow (AI \land (\text{impl}_0.bnf \sqsubseteq \text{impl}._0)) \Rightarrow (\text{spec}_0.anf \sqsubseteq \text{spec}_0)
\]

where

\[
\text{ai}_0 \equiv (AI)[\text{spec},\ \text{impl}_0]_0
\]

The proof is:

\[
\begin{align*}
\text{spec} \cdot\ af,\ spec.gf :: [pre,\ post] \\
\equiv \text{Object Specification (Semantics for Values) (6.5)} \\
\text{spec} :: [pre,\ post \land \text{spec}_0.anf \sqsubseteq \text{spec}_0.anf \land \text{spec}_0.gnf \sqsubseteq \text{spec}_0.gnf] \\
\overset{\text{REP}}{\Rightarrow} \text{Data Refine Specification (2.4)} \\
\left[ \begin{array}{c}
\text{con spec}_0 \\
\text{impl} :: \\
\end{array} \right]
\end{align*}
\]

\[
\equiv \text{Remove Logical Constant (B.45) \ and \ definition \ of \ ai} \\
\left[ \begin{array}{c}
\text{con spec}_0 \\
\text{impl} :: \\
\end{array} \right]
\end{align*}
\]
\( \equiv \) Object-Refinement Transitivity (B.65)
\[
\left[ \begin{array}{l}
\text{\hspace{1cm} con spec}_0 \bullet \\
\text{\hspace{1cm} impl:}
\end{array} \right. \\
\left. \begin{array}{l}
(\exists \text{spec} \bullet AI \land \text{spec.gf} \subseteq^{\leq} \text{impl.gf} \land \\
\text{spec.gnf} \subseteq^{\leq} \text{impl.gnf} \land \\
\text{post} \land \\
\text{pre}[\text{spec}\!\setminus\!\text{spec}_0]
\end{array} \right]
\]
\]
\( \equiv \) Strengthen Postcondition (B.54)
As \((\text{spec}_0.\text{gnf} \subseteq^{\leq} \text{impl}_0.\text{gnf}) \land (\text{impl}_0.\text{gnf} \subseteq^{\leq} \text{impl}.\text{gnf}) \Rightarrow \\
(\text{spec}_0.\text{gnf} \subseteq^{\leq} \text{impl}.\text{gnf})\)
\[
\left[ \begin{array}{l}
\text{\hspace{1cm} con spec}_0 \bullet \\
\text{\hspace{1cm} impl:}
\end{array} \right. \\
\left. \begin{array}{l}
(\exists \text{spec} \bullet AI \land \text{spec.gf} \subseteq^{\leq} \text{impl.gf} \land \\
\text{spec.gnf} \subseteq^{\leq} \text{impl.gnf} \land \\
\text{impl}_0.\text{gnf} \subseteq^{\leq} \text{impl}.\text{gnf} \land \\
\text{post} \land \\
\text{pre}[\text{spec}\!\setminus\!\text{spec}_0]
\end{array} \right]
\]
\]
\( \equiv \) Strengthen Postcondition (B.54)
Using the assumption
\(\text{AI}_0 \Rightarrow (\text{AI} \land \text{impl}_0.\text{bnf} \subseteq^{\leq} \text{impl}.\text{bnf} \Rightarrow \text{spec}_0.\text{anf} \subseteq^{\leq} \text{spec}.\text{anf})\)
\[
\left[ \begin{array}{l}
\text{\hspace{1cm} con spec}_0 \bullet \\
\text{\hspace{1cm} impl:}
\end{array} \right. \\
\left. \begin{array}{l}
(\exists \text{spec} \bullet AI \land \text{spec.gf} \subseteq^{\leq} \text{impl.gf} \land \\
\text{spec.gnf} \subseteq^{\leq} \text{impl.gnf} \land \\
\text{impl}_0.\text{bnf} \subseteq^{\leq} \text{impl}.\text{bnf} \land \\
\text{post} \land \\
\text{pre}[\text{spec}\!\setminus\!\text{spec}_0]
\end{array} \right]
\]
\]
\( \equiv \) Modified Object Specification (Semantics for Values) (7.14)
\[
\left[ \begin{array}{l}
\text{\hspace{1cm} con spec}_0 \bullet \\
\text{\hspace{1cm} impl}.\text{gf},\text{bnf}
\end{array} \right. \\
\left. \begin{array}{l}
(\exists \text{spec} \bullet AI \land \text{spec.gf} \subseteq^{\leq} \text{impl.gf} \land \\
\text{spec.gnf} \subseteq^{\leq} \text{impl.gnf} \land \\
\text{post} \land \\
\text{pre}[\text{spec}\!\setminus\!\text{spec}_0]
\end{array} \right]
\]
\]
\text{QED}

D.9 Class Proofs

Proof of 7.49 from p119 (Class Introduction)
Duplicate (Class Introduction) of 7.49 on page 119.

This refinement rule can be used to introduce a new class into an existing class hierarchy. Since a class definition is merely a scoped variable introduction with a specific object initialisation, similar rules apply to class introduction as to variable introduction. Namely, a class with class name `Classname` can be introduced into a program `P` provided `Classname` is not free in `P`:

\[
\begin{align*}
\text{Classname} &\text{ nfi } P \\
P &\sqsubseteq \left[\text{ class Classname is } \text{ Attribs end } \bullet \ P \right]
\end{align*}
\]

where `Attribs` is a list of attributes.

Using `Attribs` implicitly as both the object containing the list of attributes `Attribs` and the list of attributes:

\[
P \equiv \text{Conjunctivity} \quad w:: \left[ A(P) \ , \ E(P) \right] \quad \\
\sqsubseteq \text{Introduce Local Variable Block (B.55)} \quad \left[\left[\text{ var Classname : } \tau(\text{Classname}) \bullet w, \text{Classname}:: \left[ A(P) \ , \ E(P) \right] \right]\right] \quad \\
\sqsubseteq \text{Introduce Sequential Composition (B.40)} \quad \text{Introduce Assignment (B.32)} \quad \\
\left[\left[\text{ var Classname : } \tau(\text{Classname}) \bullet \\
\text{Classname} := \text{Attribs}; w, \text{Classname}:: \left[ A(P) \ , \ E(P) \right] \right]\right] \quad \\
\equiv \text{Scoped Object Definition (Semantics for Values) (6.1)} \quad \text{Contract Frame (B.51)} \quad \\
\text{Conjunctivity} \quad \left[\left[\text{ class Classname is } \text{Attribs end } \bullet \ P \right]\right]
\]

QED
# D.10 Semantics for References Proofs

## Proof of 8.6 from p127 (Reference Specification Sequential Composition)

Duplicate (Reference Specification Sequential Composition) of 8.6 on page 127.

In an environment with variables $\bar{x}$, with predicate $\text{mid}$ containing no initial variables except $\bar{a}_0$ (and implicitly $\text{store}_0$), and disjoint variable vectors $\bar{\alpha}$ and $\bar{\beta}$, then

$$\bar{\alpha}, \bar{\beta} : [\text{pre}, \text{post}]_*$$

$$\subseteq$$

$$[[\text{con } \tilde{A}, \text{STORE } 
\bar{\alpha}, \bar{\beta} : [\text{pre}, \text{mid}]_* ;
\bar{\alpha}, \bar{\beta} : (\text{mid})[\bar{a}_0, \text{store}_0 \tilde{A}, \text{STORE}]_*] ;$$

$$\bar{\alpha}, \bar{\beta} : (\text{post})[\bar{a}_0, \text{store}_0 \tilde{A}, \text{STORE}]_*]$$

$$]]$$

$$\bar{\alpha}, \bar{\beta} : [\text{pre}, \text{post}]_*$$

$$\equiv$$ Reference Specification (8.4)

Constrained (8.5)

$\bar{\beta}$ not in frame.

$$\text{store}, \bar{\alpha} : [\text{pre}, \text{post} \land \bar{\beta} \in (\text{dom } \text{store}_0 \leftarrow \text{store}) \supseteq \bar{\beta} \in \text{store}_0]$$

$$\subseteq$$ Introduce Sequential Composition (B.40)

$$[[\text{con STORE, } \tilde{A} \bullet$$

$$\text{store}, \bar{\alpha} : [\text{pre}, \text{mid} \land \bar{\beta} \in (\text{dom } \text{store}_0 \leftarrow \text{store}) \supseteq \bar{\beta} \in \text{store}_0] ;$$

$$[\text{mid} \land \bar{\beta} \in (\text{dom } \text{store}_0 \leftarrow \text{store}) \supseteq \bar{\beta} \in \text{store}_0]$$

$$[\text{store}_0, \bar{a}_0 \setminus \text{STORE, } \tilde{A}] \supseteq$$

$$[\text{post} \land \bar{\beta} \in (\text{dom } \text{store}_0 \leftarrow \text{store}) \supseteq \bar{\beta} \in \text{store}_0]$$

$$[\text{store}_0, \bar{a}_0 \setminus \text{STORE, } \tilde{A}]$$

$$]]$$

$$\equiv$$ Substitutions

$$[[\text{con STORE, } \tilde{A} \bullet$$

$$\text{store}, \bar{\alpha} : [\text{pre}, \text{mid} \land \bar{\beta} \in (\text{dom } \text{store}_0 \leftarrow \text{store}) \supseteq \bar{\beta} \in \text{store}_0] ;$$

$$[\text{mid} \land \bar{\beta} \in (\text{dom } \text{store}_0 \leftarrow \text{store}) \supseteq \bar{\beta} \in \text{store}_0]$$

$$[\text{store}_0, \bar{a}_0 \setminus \text{STORE, } \tilde{A}] \land$$

$$\bar{\beta} \in (\text{dom } \text{STORE} \leftarrow \text{store}) \supseteq \bar{\beta} \in \text{STORE}$$

$$[\text{post} \land \bar{\beta} \in (\text{dom } \text{store}_0 \leftarrow \text{store}) \supseteq \bar{\beta} \in \text{store}_0]$$

$$[\text{store}_0, \bar{a}_0 \setminus \text{STORE, } \tilde{A}] \land$$

$$\bar{\beta} \in (\text{dom } \text{STORE} \leftarrow \text{store}) \supseteq \bar{\beta} \in \text{STORE}$$

$$]]$$
\[\equiv \text{Reference Specification (8.4)} \]

\[
\begin{align*}
\left[ \text{con } \text{STORE}, A \bullet \right] \quad \alpha, \beta; & \quad \left[ \text{pre}, \text{mid} \right] *; \\
\left[ \text{store}, \alpha; \right] & \quad \left( \text{mid}[\text{store}_0, \alpha_0 \setminus \text{STORE}, \tilde{A}] \land \beta \iff (\text{dom}\text{STORE} \triangleleft \text{store}) \supseteq \beta \iff \text{STORE} \right) \\
\left[ \text{post}[\text{store}_0, \alpha_0 \setminus \text{STORE}, \tilde{A}] \land \beta \iff (\text{dom}\text{STORE} \triangleleft \text{store}) \supseteq \beta \iff \text{STORE} \right]
\end{align*}
\]

\[\equiv \text{Strengthen Postcondition (B.54)} \]


\[\equiv \text{Weaken Precondition (B.53)} \]

\[
\begin{align*}
\left[ \text{con } \text{STORE}, A \bullet \right] \quad \alpha, \beta; & \quad \left[ \text{pre}, \text{mid} \right] *; \\
\left[ \text{store}, \alpha; \right] & \quad \left( \text{mid}[\text{store}_0, \alpha_0 \setminus \text{STORE}, \tilde{A}] \right) \\
\left[ \text{post}[\text{store}_0, \alpha_0 \setminus \text{STORE}, \tilde{A}] \land \beta \iff (\text{dom}\text{STORE} \triangleleft \text{store}) \supseteq \beta \iff \text{store}_0 \right]
\end{align*}
\]

\[\equiv \text{Reference Specification (8.4)} \]

\[
\begin{align*}
\left[ \text{con } \text{STORE}, A \bullet \right] \quad \alpha, \beta; & \quad \left[ \text{pre}, \text{mid} \right] *; \\
\left[ \text{store}, \alpha; \right] & \quad \left( \text{mid}[\text{store}_0, \alpha_0 \setminus \text{STORE}, \tilde{A}] \right) \\
\left( \text{post}[\text{store}_0, \alpha_0 \setminus \text{STORE}, \tilde{A}] \land \beta \iff (\text{dom}\text{STORE} \triangleleft \text{store}) \supseteq \beta \iff \text{store}_0 \right)
\end{align*}
\]

\(\square\)

**Lemma D.6 (Reference Specification Sequential Composition Lemma)** Informally, \(\text{store}\) must be shown to be \(\text{STORE}\) possibly modified only at \(\beta\) and possibly extended. It is known that \(\text{store}_0\) is such a modification of \(\text{STORE}\). One choice then would be to strengthen \(\text{store}\) to \(\text{store}_0\). Another choice is to allow \(\text{store}\) to be a modified and extended version of \(\text{store}_0\). Given \(\beta \subseteq \text{dom}\text{STORE}\):

\[
\begin{align*}
\beta \iff (\text{dom}\text{STORE} \triangleleft \text{store}_0) & \supseteq \beta \iff \text{STORE} \\
\Rightarrow \quad \beta \iff (\text{dom}\text{STORE} \triangleleft \text{store}) & \supseteq \beta \iff \text{STORE} \\
\Leftarrow \quad \beta \iff (\text{dom}\text{store}_0 \triangleleft \text{store}) & \supseteq \beta \iff \text{store}_0
\end{align*}
\]

**Proof of 8.7 from p128 (Introduce Reference Clone Command)**
Duplicate (Introduce Reference Clone Command) of 8.7 on page 128.

For expression $\beta$,

\[
\alpha : [ \alpha =^\sigma \land \alpha \models^s \beta_0 ] \cup \alpha := \text{new } \beta
\]

where $\beta_0$ is $\beta[\text{store}, \alpha \setminus \text{store}_0, \alpha_0]$. 

\[\alpha : [ \alpha =^\sigma \land \alpha \models^s \beta_0 ] \cup \alpha := \text{new } \beta\]

$\equiv$ Aliased (8.1)

Assume $\gamma$ is the set of references not aliased to $\alpha$.

\[\alpha : [ (\alpha =^\sigma_\gamma) \land \alpha \models^s \beta_0 ] \cup \alpha := \text{new } \beta\]

$\equiv$ Reference Specification (8.4)

$\text{store, } \alpha : [ (\alpha =^\sigma_\gamma) \land \alpha \models^s \beta_0 \land \emptyset \models^\leq \text{store} ]$

$\equiv$ Introduce Local Variable Block (B.55)

\[[\text{var } t : \text{Ref} \bullet \text{store, } \alpha, t : [ (\alpha =^\sigma_\gamma) \land \alpha \models^s \beta_0 \land \emptyset \models^\leq \text{store} ] ]\]

$\equiv$ Introduce Sequential Composition (B.40)

\[[\text{var } t : \text{Ref} \bullet \text{var } t : [ (\forall i \bullet t \notin \text{dom } \text{store}_i) ] ; \text{store, } \alpha, t : [ (\forall i \bullet t \notin \text{dom } \text{store}_i) , (\alpha =^\sigma_\gamma) \land \alpha \models^s \beta_0 \land \emptyset \models^\leq \text{store} ] ]\]
\section*{Appendix D. Proofs}

\begin{itemize}
  \item Introduce Sequential Composition (B.40)
    \begin{align*}
      &\mathbf{[[}\mathbf{var}\ t : \mathbf{Ref} \bullet \mathbf{]} \mathbf{;} \\
      &\mathbf{t} : \mathbf{[(\forall\ i \bullet t \notin \text{dom store}_i) \mathbf{]};} \\
      &\mathbf{]}\mathbf{]} \mathbf{;} \\
      &\mathbf{[[}\mathbf{con}\ \mathbf{STORE}, A \bullet \mathbf{]} \mathbf{;} \\
      &\mathbf{\mathbf{store}, \alpha : \[(\forall\ i \bullet t \notin \text{dom store}_i) ; \ t^{\mathbf{=}} \wedge t^{\mathbf{\exists}^\gamma} \beta_0 \wedge \emptyset \mathbf{\cap store}] ; \\
      &\mathbf{]}\mathbf{]} \mathbf{;} \\
      &\mathbf{\mathbf{store, \alpha, t : \[(\alpha^{\mathbf{=}}^\gamma) \wedge \alpha^{\mathbf{\exists}^\gamma} \beta_0 \mathbf{\mid}_{\text{store}_0 \setminus \text{STORE}, A} \wedge \\
      &\mathbf{\emptyset \mathbf{\cap store}}_{\text{store}_0 \setminus \text{STORE}} \mathbf{]} \mathbf{];} \\
      &\mathbf{]}\mathbf{]} \end{align*}
  
  \item Constrained (8.5)
    \begin{itemize}
      \item Strengthen Postcondition (B.54) Using \((\forall\ i \bullet t \notin \text{dom store}_0 i) \Rightarrow t^{\mathbf{=}}\)
      \item Contract Frame (B.51)
    \end{itemize}
    \begin{align*}
      &\mathbf{[[}\mathbf{var}\ t : \mathbf{Ref} \bullet \mathbf{]} \mathbf{;} \\
      &\mathbf{t} : \mathbf{[(\forall\ i \bullet t \notin \text{dom store}_i) \mathbf{]};} \\
      &\mathbf{]}\mathbf{]} \mathbf{;} \\
      &\mathbf{[[}\mathbf{con}\ \mathbf{STORE}, A \bullet \mathbf{]} \mathbf{;} \\
      &\mathbf{\mathbf{store : \[(\forall\ i \bullet t \notin \text{dom store}_i) ; \ (t^{\mathbf{\exists}^\gamma} \beta_0 \mathbf{\mid}_{\text{store}_0 \setminus \text{STORE}, A} \wedge \\
      &\mathbf{(dom\ \text{STORE} \mathbf{\cap} \text{store}) \mathbf{\supseteq}\ \text{STORE})] ;} \\
      &\mathbf{]}\mathbf{]} \mathbf{;} \\
      &\mathbf{\alpha : \[(\alpha^{\mathbf{=}}^\gamma) \wedge \alpha^{\mathbf{\exists}^\gamma} \beta_0 \mathbf{\mid}_{\text{store}_0 \setminus \text{STORE}, A} \wedge \\
      &\mathbf{(dom\ \text{STORE} \mathbf{\cap} \text{store}) \mathbf{\supseteq}\ \text{STORE}) \mathbf{]} \mathbf{];} \\
      &\mathbf{]}\mathbf{]} \end{align*}
  
  \item Strengthen Postcondition (B.54) Using
    \begin{align*}
      &\mathbf{(t^{\mathbf{\exists}^\gamma} \beta_0 \mathbf{\mid}_{\text{store}_0 \setminus \text{STORE}, A} \wedge} \\
      &\mathbf{(dom\ \text{STORE} \mathbf{\cap} \text{store}) \mathbf{\supseteq}\ \text{STORE})[\alpha \setminus \alpha_0] \wedge} \\
      &\mathbf{(\alpha^{\mathbf{=}}^\gamma) \wedge \alpha = t} \\
      &\Rightarrow} \\
      &\mathbf{(\alpha^{\mathbf{=}}^\gamma) \wedge \alpha^{\mathbf{\exists}^\gamma} \beta_0 \mathbf{\mid}_{\text{store}_0 \setminus \text{STORE}, A} \wedge} \\
      &\mathbf{(dom\ \text{STORE} \mathbf{\cap} \text{store}) \mathbf{\supseteq}\ \text{STORE})}
    \end{align*}
    which holds as \(\alpha\) is not free in \(\beta_0\)
  
  \item Weaken Precondition (B.53)
  
  \item Remove Logical Constant (B.45)
    \begin{align*}
      &\mathbf{[[}\mathbf{var}\ t : \mathbf{Ref} \bullet \mathbf{]} \mathbf{;} \\
      &\mathbf{t} : \mathbf{[(\forall\ i \bullet t \notin \text{dom store}_i) \mathbf{]};} \\
      &\mathbf{]}\mathbf{]} \mathbf{;} \\
      &\mathbf{[[}\mathbf{store : \[(\forall\ i \bullet t \notin \text{dom store}_i) ; \ (t^{\mathbf{\exists}^\gamma} \beta_0 \mathbf{\mid}_{\text{store}_0 \setminus \text{STORE}, A} \wedge \\
      &\mathbf{(dom\ \text{STORE} \mathbf{\cap} \text{store}) \mathbf{\supseteq}\ \text{STORE})] ;} \\
      &\mathbf{]}\mathbf{]} \mathbf{;} \\
      &\mathbf{\alpha : \[(t^{\mathbf{=}}) \wedge \alpha^{\mathbf{=}}^\gamma \wedge \alpha = t \mathbf{]} \mathbf{];} \\
      &\mathbf{]}\mathbf{]} \end{align*}
\end{itemize}
Strengthen Postcondition (B.54) Using
\((\forall i \bullet t \not\in \text{dom store}_0) \land t \not\in \beta_0[\alpha_0] \land (\{t\} \ll (\text{dom store}_0 \cup \text{store}) \supseteq \{t\} \ll \text{store}_0) \Rightarrow (t \not\in \beta_0[\alpha_0] \land ((\text{dom store}_0 \cup \text{store}) \supseteq \text{store}_0))\)

Weaken Precondition (B.53)

\[\text{Mod Object Specification (Semantics for References) (7.15)}\]

\[\text{Simple Specification (B.52)}\]

QED

**Proof of 8.9 from p128 (Introduce Reference Clone Field Update)**

Duplicate (Introduce Reference Clone Field Update) of 8.9 on page 128.

For expression \(\beta\),

\[o^\uparrow \cdot \alpha :: [o^\uparrow \cdot \alpha \not\in \alpha \land o^\uparrow \cdot \alpha \not\in \beta_0] \subseteq o^\uparrow \cdot \alpha :: \text{new} \beta\]

where \(\beta_0\) is \(\beta[\text{store}, o \setminus \text{store}_0, \alpha_0]\).

The proof is analogous to that for Introduce Reference Clone Command (8.7). The main differences are that the rules for object specifications are used instead of those for classical specifications, e.g., Theorem Object Specification Strengthen Postcondition.
(7.19) is used instead of Theorem Strengthen Postcondition (B.54). Additionally, the rule for introducing field updates, Theorem 7.30, is used instead of that for introducing a classical assignment, Theorem B.52.

QED

Proof of 8.10 from p128 (Dereference Contract Frame)

Duplicate (Dereference Contract Frame) of 8.10 on page 128.

For disjoint $\alpha$ and $\beta$,

$$\bar{\alpha}, \bar{\beta}, \bar{\gamma} : \begin{array}{c}
pre \land \bar{\alpha} \subseteq \alpha', \\
\subseteq \bar{\beta}, \bar{\gamma} : \begin{array}{c}
pre \land post
\end{array}.
\end{array}$$

The proof relies on Definitions Reference Specification (8.4), Constrained (8.5) and Object Dereference (6.13).

QED
APPENDIX D. PROOFS

Proof of 8.11 from p131 (Unannotated Coalesced Specification)

Duplicate (Unannotated Coalesced Specification) of 8.11 on page 131.
For disjoint variable sets $\tilde{g}, \tilde{h}, \{\alpha\}$ and $\tilde{\beta}$:
\[
\alpha, \tilde{\beta}, \tilde{g}, \tilde{h} : [\text{pre} \land \alpha = \tilde{\beta}, \text{post}]_*
\]
coalesces to
\[
\alpha, \tilde{g}, \tilde{h} : [\text{pre}[\tilde{\beta}\setminus\alpha, ..., \alpha] \land \alpha = \varnothing, \text{post} [\tilde{\beta}, \tilde{\beta}_0\setminus\alpha, ..., \alpha, \alpha_0, ..., \alpha_0]]_*
\]

Using $\text{rep } p \equiv (\exists \tilde{\beta} \bullet \tilde{\beta} = \alpha, ..., \alpha \land p)$ (as specified in Section 8.4.3):
\[
\alpha, \tilde{\beta}, \tilde{g}, \tilde{h} : [\text{pre} \land \alpha = \tilde{\beta}, \text{post}]_*
\]
≡ Reference Specification (8.4)
\[
\text{store, } \alpha, \tilde{\beta}, \tilde{g} : [\text{pre} \land \alpha = \tilde{\beta}, \text{post} \land \tilde{h}_0 \in \text{store}]
\]
≡ Initial Variable (B.49)
\[
[[ \text{con } \text{STORE, } \tilde{H}, \tilde{G}, A, \tilde{B} \bullet \\
\text{store, } \alpha, \tilde{\beta}, \tilde{g} : [\text{pre} \land \alpha = \tilde{\beta} \land \text{STORE} = \text{store} \land A = \alpha \land \text{H} = \tilde{h} \land \text{G} = \tilde{g} \land \tilde{B} = \tilde{\beta} \\
\text{post} \land \tilde{h}_0 \in \text{store}] \\
[\text{store}_0, \tilde{h}_0, \tilde{g}_0, \alpha_0, \tilde{\beta}_0 \setminus \text{STORE, } \tilde{H}, \tilde{G}, A, \tilde{B}]]]
\]
\leq_{\text{rep}} \text{ Data Refine Specification (2.4)}
\[
[[ \text{con } \text{STORE, } \tilde{H}, \tilde{G}, A, \tilde{B} \bullet \\
\text{store, } \alpha, \tilde{\beta}, \tilde{g} : [\exists \tilde{\beta} \bullet \tilde{\beta} = \alpha, ..., \alpha \land \text{pre} \land \alpha = \tilde{\beta} \land \text{STORE} = \text{store} \land A = \alpha \land \text{H} = \tilde{h} \land \text{G} = \tilde{g} \land \tilde{B} = \tilde{\beta} \\
\text{post} \land \tilde{h}_0 \in \text{store}] \\
[\text{store}_0, \tilde{h}_0, \tilde{g}_0, \alpha_0, \tilde{\beta}_0 \setminus \text{STORE, } \tilde{H}, \tilde{G}, A, \tilde{B}]]]
\]
≡ Existential Quantification One Point Rule (B.6)
\[
[[ \text{con } \text{STORE, } \tilde{H}, \tilde{G}, A, \tilde{B} \bullet \\
\text{store, } \alpha, \tilde{\beta}, \tilde{g} : [\text{pre}[\tilde{\beta}\setminus\alpha, ..., \alpha] \land \alpha = \varnothing \land \text{STORE} = \text{store} \land A = \alpha \land \text{H} = \tilde{h} \land \text{G} = \tilde{g} \land \tilde{B} = \alpha, ..., \alpha \\
\text{post} \land \tilde{h}_0 \in \text{store}] \\
[\text{store}_0, \tilde{h}_0, \tilde{g}_0, \alpha_0, \tilde{\beta}_0 \setminus \text{STORE, } \tilde{H}, \tilde{G}, A, \tilde{B}][\tilde{\beta}\setminus\alpha, ..., \alpha]]]
\]
\[\equiv \text{Strengthen Postcondition (B.54)}
\]
\[\text{Weaken Precondition (B.53)}
\]
\[\begin{align*}
\text{store, } \alpha, \tilde{g} : & \quad \left[ \begin{array}{c}
\text{pre} [\tilde{\beta} \setminus \alpha, \ldots, \alpha] \land \alpha = \emptyset \\
\text{(post} \land \tilde{h}_0 \in \text{store})
\end{array} \right] \\
\text{store}_0, \tilde{h}_0, \tilde{g}_0, \alpha_0, \tilde{\beta}_0 \setminus \text{STORE, } \tilde{H}, \tilde{G}, \tilde{A}, \tilde{B}
\end{align*}\]
\[\left[ \begin{array}{c}
\text{post} [\tilde{\beta}_0 \setminus \alpha_0, \ldots, \alpha_0] [\tilde{\beta} \setminus \alpha, \ldots, \alpha] [\text{STORE, } \tilde{A}, \tilde{H}, \tilde{G}, \tilde{B} \setminus \text{store}_0, \alpha_0, \tilde{h}_0, \tilde{g}_0, \alpha_0, \ldots, \alpha_0]
\end{array} \right]
\]\
\[\equiv \text{Substitutions}
\]
\[\text{Remove Logical Constant (B.45)}
\]
\[\begin{align*}
\text{store, } \alpha, \tilde{g} : & \quad \left[ \begin{array}{c}
\text{pre} [\tilde{\beta} \setminus \alpha, \ldots, \alpha] \land \alpha = \emptyset \\
\text{(post} \land \tilde{h}_0 \in \text{store})
\end{array} \right] \\
\text{store}_0, \tilde{h}_0, \tilde{g}_0, \alpha_0, \tilde{\beta}_0 \setminus \text{STORE}
\end{align*}\]
\[\left[ \begin{array}{c}
\text{post} [\tilde{\beta}_0 \setminus \alpha_0, \ldots, \alpha_0] [\tilde{\beta} \setminus \alpha, \ldots, \alpha] \\
\alpha_0, \tilde{h}_0, \tilde{g}_0, \alpha_0, \ldots, \alpha_0
\end{array} \right]
\]\
\[\equiv \text{Reference Specification (8.4)}
\]
\[\begin{align*}
\alpha, \tilde{g}, \tilde{h} : & \quad \left[ \begin{array}{c}
\text{pre} [\tilde{\beta} \setminus \alpha, \ldots, \alpha] \land \alpha = \emptyset \\
\text{post} [\tilde{\beta}_0 \setminus \alpha_0, \ldots, \alpha_0] [\tilde{\beta} \setminus \alpha, \ldots, \alpha, \alpha_0, \ldots, \alpha_0]
\end{array} \right]
\end{align*}\]
\[QED
\]

**Proof of 8.12 from p132 (Dereferenced Coalesced Specification)**

**Duplicate (Dereferenced Coalesced Specification) of 8.12 on page 132.**

For disjoint variable sets \( \tilde{g}, \tilde{h}, \left\{ \alpha \right\} \) and \( \tilde{\beta} \):

\[
\left[ \begin{array}{c}
\text{pre} \land \alpha = \emptyset \\
\text{post}
\end{array} \right]
\]

\[
\alpha, \tilde{g}, \tilde{h} : \quad \left[ \begin{array}{c}
\text{pre} [\tilde{\beta} \setminus \alpha, \ldots, \alpha] \land \alpha = \emptyset \\
\text{post} [\tilde{\beta}_0 \setminus \alpha_0, \ldots, \alpha_0] [\tilde{\beta} \setminus \alpha, \ldots, \alpha, \alpha_0, \ldots, \alpha_0]
\end{array} \right]
\]

Using \( \text{rep } p \equiv (\exists \tilde{\beta} \bullet \tilde{\beta} = \alpha, \ldots, \alpha \land p) \):

\[
\left[ \begin{array}{c}
\text{pre} \land \alpha = \emptyset \\
\text{post}
\end{array} \right]
\]

\[\equiv \text{Reference Specification (8.4)}
\]
\[\begin{align*}
\text{store}, \tilde{g} : & \quad \left[ \begin{array}{c}
\text{pre} \land \alpha = \emptyset \\
\text{post} \land (\tilde{h}_0 \cup \alpha_0 \cup \tilde{\beta}_0) \in \text{store}
\end{array} \right]
\end{align*}\]
\[\left[ \begin{array}{c}
\text{Data Refine Specification (2.4)}
\end{array} \right]
\]
\[\left[ \begin{array}{c}
\text{con } \tilde{B}, \text{STORE, } G, \tilde{\beta} \bullet
\end{array} \right]
\]
\[\left[ \begin{array}{c}
\left( \exists \tilde{\beta} \bullet \tilde{\beta} = \alpha, \ldots, \alpha \land \\
\text{post} \land (\tilde{h}_0 \cup \alpha_0 \cup \tilde{\beta}_0) \in \text{store}ight) [\tilde{\beta}_0, \text{store}_0, \tilde{g}_0 \setminus \tilde{B}, \text{STORE, } G]
\end{array} \right]
\]
\[ \begin{align*}
\quad & \text{Strengthen Postcondition (B.54)} \\
\quad & \text{Remove Logical Constant (B.45)} \\
\quad & \left[ \begin{array}{c}
\text{store, } \bar{g}: \\
\quad \left[ \begin{array}{c}
\bar{B} = \alpha, \ldots, \alpha \land \text{pre}\left[\bar{\beta} \setminus \alpha, \ldots, \alpha\right] \land \alpha =^\varnothing \\
\quad \left( \exists \bar{\beta} \cdot \bar{\beta} = \alpha, \ldots, \alpha \land \text{post}\left[\bar{\beta}_0 \setminus \alpha_0, \ldots, \alpha_0\right] \land (\bar{h}_0 \cup \alpha_0 \cup \bar{B}) \subseteq \text{store} \right) \\
\end{array} \right] \\
\end{array} \right] \\
\quad & \text{Strengthen Postcondition (B.54)} \\
\quad & \text{Weaken Precondition (B.53)} \\
\quad & \text{Remove Logical Constant (B.45)} \\
\quad & \left[ \begin{array}{c}
\text{store, } \bar{g}: \\
\quad \left[ \begin{array}{c}
\text{pre}\left[\bar{\beta} \setminus \alpha, \ldots, \alpha\right] \land \alpha =^\varnothing \\
\quad \left( \text{post}\left[\bar{\beta}_0 \setminus \alpha_0, \ldots, \alpha_0, \alpha, \ldots, \alpha\right] \land (\bar{h}_0 \cup \alpha_0) \subseteq \text{store} \right) \\
\end{array} \right] \\
\end{array} \right] \\
\quad & \text{Reference Specification (8.4)} \\
\quad & \left[ \begin{array}{c}
\alpha \upharpoonright, \bar{g}, \bar{h}: \\
\quad \left[ \begin{array}{c}
\text{pre}\left[\bar{\beta} \setminus \alpha, \ldots, \alpha\right] \land \alpha =^\varnothing, \ (\text{post})\left[\bar{\beta}_0 \setminus \alpha_0, \ldots, \alpha_0, \alpha, \ldots, \alpha\right] \\
\end{array} \right] \\
\end{array} \right]
\end{align*} \]

\[ QED \]

**Proof of 8.14 from p132 (Dual Coalesced Specification)**

\[ \begin{align*}
\text{Duplicate (Dual Coalesced Specification) of 8.14 on page 132.} \\
\quad & \text{For disjoint variable sets } \bar{g}, \bar{h}, \{\alpha\} \text{ and } \bar{\beta}: \\
\quad & \alpha, \alpha \upharpoonright, \bar{\beta}, \bar{\beta}, \bar{g}, \bar{h}: \left[ \begin{array}{c}
\text{pre} \land \alpha =^\varnothing, \ \text{post} \\
\end{array} \right] \\
\quad & \text{coalesces to} \\
\quad & \alpha, \alpha \upharpoonright, \bar{g}, \bar{h}: \left[ \begin{array}{c}
\text{pre}\left[\bar{\beta} \setminus \alpha, \ldots, \alpha\right] \land \alpha =^\varnothing, \ \text{post}\left[\bar{\beta}_0 \setminus \alpha_0, \ldots, \alpha_0, \alpha, \ldots, \alpha\right] \\
\end{array} \right]
\end{align*} \]

The proof is almost identical to that for Theorem Dereferenced Coalesced Specification (8.12).

\[ QED \]
APPENDIX D. PROOFS

Proof of 8.15 from p133 (Assignments to the Primary)

Duplicate (Assignments to the Primary) of 8.15 on page 133.
For assignments to primary variable \( a \) the following transformation is applicable.

\[
\alpha := X; \\
\text{uncoalesces-to} \\
\alpha := X; \beta := \alpha, \ldots, \alpha
\]

This proof uses the \( rep' \ q \equiv q \land \beta = \alpha, \ldots, \alpha \) as presented in Section 8.4.3. This law is similar to the augment assignment rule presented by Morgan [Mor94, Law 17.8].

Given \( p \) in the coalesced environment:

\[
(rep'; \alpha := X) \ p \\
\equiv \text{Assignment (B.30)} \\
\quad rep' \ (p[\alpha \setminus X]) \\
\equiv \text{Defn } rep' \\
\quad p[\alpha \setminus X] \land \beta = \alpha, \ldots, \alpha \\
\Rightarrow \\
\quad p[\alpha \setminus X]
\]

It is recommended that the reader now start from the bottom and read to this point as the development was constructed in this manner.

\[
\equiv \text{Assignment} \\
\quad (\alpha := X) \ (p) \\
\equiv \\
\quad (\alpha := X) \ (p \land \alpha = \alpha) \\
\equiv \text{No } \beta \text{ in coalesced environment.} \\
\quad (\alpha := X) \ ((p \land \beta = \alpha, \ldots, \alpha) [\beta \setminus \alpha, \ldots, \alpha]) \\
\equiv \text{Sequential Composition and Assignment} \\
\quad (\alpha := X; \beta := \alpha, \ldots, \alpha) \ (p \land \beta = \alpha, \ldots, \alpha) \\
\equiv \text{Defn } rep' \\
\quad (\alpha := X; \beta := \alpha, \ldots, \alpha; rep') \ p
\]

QED

Proof of 8.16 from p133 (Unannotated Uncoalesced Specification)
Duplicate (Unannotated Uncoalesced Specification) of 8.16 on page 133.
When uncoalescing a specification, the coalesced variables (\(\tilde{\beta}\)) are returned to the frame and the postcondition is strengthened to ensure that subsequent code can rely on the aliasing of \(\alpha\) with \(\tilde{\beta}\).

\[
\alpha: \left[ \text{pre} \land \alpha = \emptyset, \text{post} \right],
\]

uncoalesces to

\[
\alpha, \tilde{\beta}: \left[ \alpha = \tilde{\beta} \land \text{pre}, \alpha = \tilde{\beta} \land \text{post} \right],
\]

This proof uses the \(\text{rep'} q \equiv q \land \tilde{\beta} = \alpha, ..., \alpha\) as presented in Section 8.4.3.

\[
\alpha: \left[ \text{pre} \land \alpha = \emptyset, \text{post} \right],
\]

\[\equiv\] Reference Specification (8.4)
\[
\alpha: \left[ \text{pre} \land \alpha = \emptyset, \text{post} \right],
\]

\[\equiv\] Initial Variable (B.49)

\[
\left[ \text{con A} \bullet \right]
\]

\[
\alpha: \left[ \text{pre} \land \alpha = \emptyset \land \alpha = A, \text{post}[\alpha_0 \setminus A] \right],
\]

\[\preceq_{\text{rep'}}\] Data Refine Specification (2.4)

\[
\left[ \text{con A} \bullet \right]
\]

\[
\alpha, \tilde{\beta}: \left[ \tilde{\beta} = \alpha, ..., \alpha \land \text{pre} \land \alpha = \tilde{\beta} \land \alpha = A, \tilde{\beta} = \alpha, ..., \alpha \land \text{post}[\alpha_0 \setminus A] \right],
\]

\[\equiv\] Strengthen Postcondition (B.54) with \(\alpha_0 = A\)

Weaken Precondition (B.53)

Remove Logical Constant (B.45)

\[
\alpha, \tilde{\beta}: \left[ \text{pre} \land \alpha = \tilde{\beta}, \alpha = \tilde{\beta} \land \text{post} \right],
\]

\[\equiv\] Reference Specification (8.4)

\[
\alpha, \tilde{\beta}: \left[ \alpha = \tilde{\beta} \land \text{pre}, \alpha = \tilde{\beta} \land \text{post} \right],
\]

QED

Proof of 8.17 from p133 (Dereferenced Uncoalesced Specification)
Duplicate (Dereferenced Uncoalesced Specification) of 8.17 on page 133.

For specifications involving a frame with a dereferenced primary:

\[ \alpha : [\text{pre} \land \alpha^= \land \text{post}]. \]

uncoalesces-to

\[ \alpha^\dagger, \beta^\dagger : [\alpha^= \land \text{pre} \land \text{post}]. \]

The strengthening of the postcondition is not required as the frame annotation of \( \alpha \) prevents the reference \( \alpha \) from being altered.

This proof uses the \( \text{rep'} q \cong q \land \beta = \alpha, ..., \alpha \) as presented in Section 8.4.3.

\[ \alpha^\dagger : [\text{pre} \land \alpha^= \land \text{post}]. \]

\[ \equiv \text{Reference Specification (8.4)} \]

\[ \text{store} : [\text{pre} \land \alpha^= \land \text{post} \land \{\alpha_0\} \langle \text{store} \rangle [\text{store} \setminus \text{STORE}]] \]

Using no specification variables, and hence no specification variables in the frame, common frame variable \( \text{store} \) and implementation variables \( \beta \):

\[ \preceq_{\text{rep'}} \text{Data Refine Specification (2.4)} \]

\[ \llbracket \text{con STORE} \rrbracket \]

\[ \beta, \text{store} : \left[ \beta = \alpha, ..., \alpha \land \text{pre} \land \alpha^= \land \text{STORE} = \text{store} \right] \]

\[ \beta = \alpha, ..., \alpha \land (\text{post} \land \{\alpha_0\} \langle \text{store} \rangle [\text{store} \setminus \text{STORE}]) \]

\[ \equiv \text{Strengthen Postcondition (B.54)} \] with

\[ \text{STORE} = \text{store}_0 \] and using

\[ \beta_0 = \beta \land \beta_0 = \alpha, ..., \alpha \Rightarrow \beta = \alpha, ..., \alpha \]

\[ \equiv \text{Weaken Precondition (B.53)} \]

Remove Logical Constant (B.45)

\[ \beta, \text{store} : \left[ \alpha^= \land \text{pre} \land \beta_0 = \beta \land \text{post} \land \{\alpha_0\} \langle \text{store} \rangle \right] \]

\[ \equiv \text{Expand Frame (B.50)} \]

\[ \beta, \text{store}, \alpha : \left[ \alpha^= \land \text{pre} \land \alpha = \alpha_0 \land \beta_0 = \beta \land \text{post} \land \{\alpha_0\} \langle \text{store} \rangle \right] \]

\[ \equiv \text{Strengthen Postcondition (B.54)} \]

Using \( \beta_0 = \alpha_0, ..., \alpha_0 \Rightarrow \{\alpha_0\} = \{\alpha_0, \beta_0\} \)

\[ \beta, \text{store}, \alpha : \left[ \alpha^= \land \text{pre} \land \alpha = \alpha_0 \land \beta_0 = \beta \land \text{post} \land \{\alpha_0, \beta_0\} \langle \text{store} \rangle \right] \]

\[ \equiv \text{Expand Frame (B.50)} \] in reverse.

\[ \text{store} : \left[ \alpha^= \land \text{pre} \land \text{post} \land \{\alpha_0, \beta_0\} \langle \text{store} \rangle \right] \]

\[ \equiv \text{Reference Specification (8.4)} \]

\[ \alpha^\dagger, \beta^\dagger : [\alpha^= \land \text{pre} \land \text{post}]. \]
APPENDIX D. PROOFS

QED

Proof of 8.18 from p133 (Dual Uncoalesced Specification)

Duplicate (Dual Uncoalesced Specification) of 8.18 on page 133.
For disjoint variable sets \( \bar{g}, \bar{h}, \{\alpha\} \) and \( \bar{\beta} \):

\[
\alpha, \alpha^\dagger, \bar{g}, \bar{h} : [\text{pre} \land \alpha = \emptyset, \text{post}]^* \]

uncoalesces-to

\[
\alpha, \alpha^\dagger, \bar{\beta}, \bar{\beta}^\dagger, \bar{g}, \bar{h} : \left[ \alpha = \bar{\beta} \land \text{pre} \land \alpha = \bar{\beta} \land \text{post} \right]^* \]

Using \( \text{rep}' \ q \equiv q \land \bar{\beta} = \alpha, \ldots, \alpha \):

\[
\alpha, \alpha^\dagger, \bar{g}, \bar{h} : [\text{pre} \land \alpha = \emptyset, \text{post}]^* \]

\equiv \text{Reference Specification (8.4)}

\[
\alpha, \text{store}, \bar{g} : [\text{pre} \land \alpha = \emptyset, \text{post} \land (\{\alpha_0\} \cup \bar{h}_0) \subseteq \text{store}] \]

\\leq_{\text{rep}'} \text{Data Refine Specification (2.4)}

\[
[\text{con } A, \text{STORE}, G \bullet \left[ \begin{array}{c}
\text{pre} \land \alpha = \bar{\beta} \land A = \alpha \land \text{STORE} = \text{store} \land G = \bar{g} \\
\alpha = \bar{\beta} \\
\text{post} \land (\{\alpha_0\} \cup \bar{h}_0) \subseteq \text{store} \end{array} \right] \text{[store}_0, g_0 \setminus A, \text{STORE}, G] ]
\]

\equiv \text{Strengthen Postcondition (B.54)}

Weaken Precondition (B.53)

Remove Logical Constant (B.45)

\[
\bar{\beta}, \alpha, \text{store}, \bar{g} : \left[ \alpha = \bar{\beta} \land \text{pre} \land \alpha = \bar{\beta} \land \text{post} \land (\{\alpha_0\} \cup \bar{\beta}_0 \cup \bar{h}_0) \subseteq \text{store} \right]
\]

\equiv \text{Reference Specification (8.4)}

\[
\alpha, \alpha^\dagger, \bar{\beta}, \bar{\beta}^\dagger, \bar{g}, \bar{h} : \left[ \text{pre} \land \alpha = \bar{\beta} \land \text{post} \land \alpha = \bar{\beta} \right]^* \]

QED

Proof of 8.20 from p135 (Inversed reps)

Duplicate (Inversed reps) of 8.20 on page 135.
Showing that coalesced programming is an instantiation of the ‘data refinement via inverse commands’ technique requires proving that \( \text{rep}' \) is an inverse of \( \text{rep} \) as shown in Section 8.3. That is,

\[
\text{skip} \equiv \text{rep}; \text{rep}'
\]

and hence, as a corollary:

\[
\text{skip} \subseteq \text{rep}; \text{rep}' \land \text{rep}; \text{rep}' \subseteq \text{skip}
\]
The proof uses $p$ in the coalesced environment. It can therefore be assumed that there is no $\tilde{\beta}$ in $p$.

$$\text{skip } p$$

$$\equiv$$

$$p$$

$$\equiv \text{No } \tilde{\beta} \text{ in } p.$$  

$$p[\tilde{\beta}\setminus\alpha]$$

$$\equiv \text{Existential Quantification One Point Rule (B.6)}$$

$$(\exists \beta \cdot p \land \beta = \alpha)$$

$$\equiv \text{Choice of } rep$$

$$rep (p \land \beta = \alpha)$$

$$\equiv \text{Choice of } rep'$$

$$(rep; \ rep')p$$

$QED$

Proof of 8.21 from p135 (Assignments to Non-Primaries)

Duplicate (Assignments to Non-Primaries) of 8.21 on page 135.

Given a variable $\gamma$, which is not the primary variable (i.e., $\alpha$),

$$\gamma := X$$

uncoalesces-to

$$\gamma := X$$

Although $X$ may contain $\alpha$, it data refines to itself as $\alpha$ is not removed by the data refinement. $X$ cannot contain $\tilde{\beta}$ as $\tilde{\beta}$ is fresh. Consequently, $rep$ acts as an identify function.

$QED$
Proof of 8.22 from p136 (Transforming Reference Specification)

Duplicate (Transforming Reference Specification) of 8.22 on page 136.
Transforming a reference specification with unaliased variables $\bar{\alpha}$ and $\bar{\beta}$ involves replacing all dereferences of $\bar{\alpha}$ and $\bar{\beta}$ with direct variable accesses. Given variables $\bar{\alpha}: \text{Ref} \{\Delta\} \text{ and } \bar{\beta}: \text{Ref} \{\Xi\}$, the following specification

$$\bar{\alpha}'[: \alpha = \emptyset \wedge \beta = \emptyset \wedge \text{pre}[\bar{\alpha}, \bar{\beta}\backslash \alpha], \bar{\beta}'] \wedge \text{post}[\bar{\alpha}, \bar{\alpha}_0, \bar{\beta}, \bar{\beta}_0\backslash \alpha], \bar{\beta}_0 \backslash \alpha, \bar{\beta}_0 \backslash \beta, \bar{\beta}_0 \backslash \beta_0] \}.$$

codes-as (with respect to $\bar{\alpha}$ and $\bar{\beta}$)

$$\bar{\alpha}'[: \text{pre}, \text{post}].$$

where $\bar{\alpha} : \Delta$ and $\bar{\beta} : \Xi$.

Using the abbreviations $\bar{z} = \bar{\alpha} \cup \bar{\beta}$ and $\bar{\bar{\alpha}} = \bar{\alpha}' \cup \bar{\beta}'$ this data refines using the initial environment variables $\alpha = \bar{\alpha} \cup \bar{\beta} \cup \{\text{store}\}$, frame elements that are initial variables $\delta = \text{store}$, common frame elements $\gamma = \emptyset$, and mirror environment variables $\beta = \bar{\alpha}' \cup \bar{\beta}' \cup \text{store}'$ and with $\text{rep}$ chosen as discussed in Section 8.5.3:

$$\text{rep } p \equiv \exists \bar{\bar{\alpha}}, \text{store} \bullet p \wedge \bar{\bar{\alpha}} = \bar{\beta}' \wedge \bar{\bar{\alpha}} \leq \text{store} = \text{store}' \wedge \bar{\bar{\alpha}} = \emptyset$$

That is:

$$\leq_{\text{rep}} \text{ Data Refine Specification (2.4)} \left[ \left[ \text{con } \text{STORE}, \bar{\alpha}, \bar{\beta}, \text{store} \bullet \right. \right.$$

$$\bar{\alpha}', \bar{\beta}', \text{store}' : \left[ \bar{\bar{\alpha}} = \bar{\beta}' \wedge \bar{\bar{\alpha}} \leq \text{store} = \text{store}' \wedge \bar{\bar{\alpha}} = \emptyset \wedge \bar{\bar{\alpha}} = \emptyset \wedge \text{pre}[\bar{\bar{\alpha}}] \wedge \text{STORE} = \text{store} \right.$$
\equiv \text{Reference Specification (8.4)}
\begin{align*}
\langle \text{con } &\text{ STORE}, \tilde{\alpha}, \tilde{\beta}, \text{store } \cdot \rangle \\
\tilde{\alpha}', \tilde{\beta}', \text{store}': \\
& \begin{cases}
\tilde{z} = \tilde{\tilde{z}} \land \tilde{z} \ll \text{store} = \text{store}' \land \\
\tilde{z}^{-\omega} \land \text{pre}[\tilde{z}, \tilde{z}] \land \text{STORE} = \text{store} \\
(\exists \text{ store } \bullet \tilde{z} = \tilde{\tilde{z}} \land \tilde{z} \ll \text{store} = \text{store}' \land \tilde{z}^{-\omega} \land \\
(\text{post}[\tilde{z}, \tilde{z_0}, \tilde{z_1}], \tilde{z_0})][\text{store}_0 \setminus \text{STORE}] \land \\
\tilde{\alpha} \ll (\text{dom store} \ll \text{store}) \exists^c \tilde{\alpha} \ll \text{STORE})
\end{cases}
\end{align*}
\]

The current goal is now to strengthen the postcondition so that \text{store}' only alters at new indices.

To make the proof more readable, the \text{store} logical constant is renamed \text{store}_0 to signify it represents the original value of the initial environment variable \text{store}.

\begin{align*}
\equiv \langle \text{con } &\text{ STORE}, \tilde{\alpha}, \tilde{\beta}, \text{store}_0 \cdot \rangle \\
\tilde{\alpha}', \tilde{\beta}', \text{store}': \\
& \begin{cases}
\tilde{z} = \tilde{\tilde{z}} \land \tilde{z} \ll \text{store}_0 = \text{store}' \land \\
\tilde{z}^{-\omega} \land \text{pre}[\tilde{z}, \tilde{z}] [\text{store}_0 \setminus \text{store}] \land \text{STORE} = \text{store}_0 \\
(\exists \text{ store } \bullet \tilde{z} = \tilde{\tilde{z}} \land \tilde{z} \ll \text{store} = \text{store}' \land \tilde{z}^{-\omega} \land \\
(\text{post}[\tilde{z}, \tilde{z_0}, \tilde{z_1}], \tilde{z_0})][\text{store}_0 \setminus \text{STORE}] \land \\
\tilde{\alpha} \ll (\text{dom store}_0 \ll \text{store}) \exists^c \tilde{\alpha} \ll \text{STORE})
\end{cases}
\end{align*}

The postcondition is strengthened using \text{STORE} = \text{store}_0 from the precondition. Subsequently, \text{STORE} is removed from the precondition and the logical constant frame.

\equiv \text{Strengthen Postcondition (B.54)}
\text{Weaken Precondition (B.53)}
\text{Remove Logical Constant (B.45)}
\begin{align*}
\langle \text{con } &\tilde{\alpha}, \tilde{\beta}, \text{store}_0 \rangle \\
\tilde{\alpha}', \tilde{\beta}', \text{store}': \\
& \begin{cases}
\tilde{\tilde{z}} = \tilde{\tilde{z}} \land \tilde{z} \ll \text{store}_0 = \text{store}' \land \\
\tilde{z}^{-\omega} \land \text{pre}[\tilde{z}, \tilde{z}] [\text{store}_0 \setminus \text{store}] \\
(\exists \text{ store } \bullet \tilde{z} = \tilde{\tilde{z}} \land \tilde{z} \ll \text{store} = \text{store}' \land \tilde{z}^{-\omega} \land \\
\text{post}[\tilde{z}, \tilde{z_0}, \tilde{z_1}], \tilde{z_0})][\text{store}_0 \setminus \text{STORE}] \land \\
\tilde{\alpha} \ll (\text{dom store}_0 \ll \text{store}) \exists^c \tilde{\alpha} \ll \text{store}_0)
\end{cases}
\end{align*}

Using \tilde{z} \ll \text{store}_0 = \text{store}' from the precondition, and \tilde{z} \ll \text{store} = \text{store}' the postcondition is modified considering the following equivalence.

\begin{align*}
(\text{dom } &\text{store}_0 \ll \text{store}') \exists^c \text{store}_0 \land \tilde{\beta} \ll \text{store} \exists^c \tilde{\beta} \ll \text{store}_0 \\
\equiv & \text{Using } \tilde{z} \ll \text{store}_0 = \text{store}'.
\end{align*}

\begin{align*}
(\text{dom } &\text{store}_0 \ll \text{store}') \exists^c \text{store}_0 \land \tilde{\beta} \ll \text{store} \exists^c \tilde{\beta} \ll \text{store}_0
\end{align*}
\[\equiv \text{Since } (\text{dom } c) \setminus b \equiv \text{dom } (b \triangleleft c).\]
\[(((\text{dom } \text{store})_0 \setminus \bar{z}) \triangleleft \text{store}') \supset \exists^c \text{store}'_0 \land \bar{\beta} \triangleleft \text{store} \equiv^c \bar{\beta} \triangleleft \text{store}_0\]
\[\equiv \text{Since } (b \setminus c) \triangleleft d = b \triangleleft (c \triangleleft d).\]
\[\text{dom } \text{store}_0 \triangleleft (\bar{z} \triangleleft \text{store}') \supset \exists^c \text{store}'_0 \land \bar{\beta} \triangleleft \text{store} \equiv^c \bar{\beta} \triangleleft \text{store}_0\]

\[\equiv \text{Using } \text{store}' = \bar{z} \triangleleft \text{store}.\]
\[\text{dom } \text{store}_0 \triangleleft (\bar{z} \triangleleft \text{store}') \supset \exists^c \text{store}'_0 \land \bar{\beta} \triangleleft \text{store} \equiv^c \bar{\beta} \triangleleft \text{store}_0\]
\[\equiv \text{Using } d \triangleleft (b \triangleleft c) \equiv b \triangleleft (d \triangleleft c).\]
\[\bar{z} \triangleleft \text{dom } \text{store}_0 \triangleleft \text{store} \supset \exists^c \text{store}'_0 \land \bar{\beta} \triangleleft \text{store} \equiv^c \bar{\beta} \triangleleft \text{store}_0\]
\[\equiv \text{Using } \text{store}'_0 = \bar{z} \triangleleft \text{store}_0.\]
\[\bar{z} \triangleleft \text{dom } \text{store}_0 \triangleleft \text{store} \supset \exists^c \bar{z} \triangleleft \text{store}_0 \land \bar{\beta} \triangleleft \text{store} \equiv^c \bar{\beta} \triangleleft \text{store}_0\]
\[= (\text{dom } \text{store}_0 \setminus \bar{z}) \triangleleft \text{store} \equiv^c (\text{dom } \text{store}_0 \setminus \bar{z}) \triangleleft \text{store}_0 \land \bar{\beta} \triangleleft \text{store} \equiv^c \bar{\beta} \triangleleft \text{store}_0\]

Using \((b \triangleleft w = b \triangleleft u \land c \triangleleft w = c \triangleleft u) \equiv (b \cup c) \triangleleft w = (b \cup c) \triangleleft u\)
\[\equiv \]
\[(((\text{dom } \text{store}_0 \setminus \bar{z}) \cup \bar{\beta}) \triangleleft \text{store} \equiv^c ((\text{dom } \text{store}_0 \setminus \bar{z}) \cup \bar{\beta}) \triangleleft \text{store}_0\]
\[\equiv \text{Using } ((c \cup d) \setminus e) \cup e = c \setminus d.\]
\[\text{dom } \text{store}_0 \setminus \bar{\alpha} \triangleleft \text{store} \equiv^c \text{dom } \text{store}_0 \setminus \bar{\alpha} \triangleleft \text{store}_0\]
\[\equiv \text{Simplifying}\]
\[\bar{\alpha} \triangleleft \text{dom } \text{store}_0 \triangleleft \text{store} \equiv^c \bar{\alpha} \triangleleft \text{store}_0\]

Consequently the previous specification refines as follows.

\[\equiv \text{Strengthen Postcondition (B.54)}\]
\[\boxed{[\text{con } \bar{\alpha}, \bar{\beta}, \text{store}_0 \bullet \begin{array}{l}
\begin{array}{c}
\bar{z}_0^1 = \bar{z} \land \bar{z} \triangleleft \text{store}_0 = \text{store}' \land \\
\bar{z}^\ominus \land \text{pre}[\bar{z}^\ominus \setminus \text{store}_0] \setminus \text{store}_0
\end{array}
\end{array} \\
\bar{z}^\ominus \land \text{pre}[\bar{z}^\ominus \setminus \text{store}_0] \setminus \text{store}_0 \setminus \text{store}_0
\end{array}]
\]
\[\equiv \text{To transform the postcondition closer to that desired, substitutions are performed on } post \text{ using } \bar{z} = \bar{z}'.\]

\[\equiv \boxed{[\text{con } \bar{\alpha}, \bar{\beta}, \text{store}_0 \bullet \begin{array}{l}
\begin{array}{c}
\bar{z}_0^1 = \bar{z} \land \bar{z} \triangleleft \text{store}_0 = \text{store}' \land \\
\bar{z}^\ominus \land \text{pre}[\bar{z}^\ominus \setminus \text{store}_0] \setminus \text{store}_0
\end{array}
\end{array} \\
\exists^c \text{store}'_0 \setminus \text{store}_0 \land \bar{\beta} \triangleleft \text{store} \equiv^c \bar{\beta} \triangleleft \text{store}_0
\end{array}]
\]
≡ Splitting conjuncts

\\[
\begin{array}{c}
\alpha', \beta'; \text{store}' : \\
\end{array}
\]

\[
\begin{array}{c}
\exists \text{store} \bullet \alpha = \alpha' \land \beta = \beta' \land \text{z} \lhd \text{store} = \text{store}' \land \text{z}^{-\ominus} \land \\
\text{dom store}' \sqsubset \text{store}' \Leftrightarrow \text{store}' \land \beta' \sqsubset \text{store}' \Leftrightarrow \exists \text{store}' \sqsubset \text{store}' \land \beta' \sqsubset \text{store}'
\end{array}
\]

Choose store \equiv \{ \alpha \mapsto \alpha' \} \cup \{ \beta \mapsto \beta' \} \cup \text{store}'. This is valid as neither \alpha \lor \beta \lor \text{store}'. There are no dereferences of \text{z} in post[\text{z}, \text{store}'] \sqcup \text{store}']\sqcup \text{store}'][0].

\square Existential Quantification One Point Rule (B.6)

\\[
\begin{array}{c}
\exists \alpha', \beta'; \text{store}' : \\
\end{array}
\]

\[
\begin{array}{c}
\exists \text{store} \bullet \alpha = \alpha' \land \beta = \beta' \land \text{store} = \text{store}' \land \text{z}^{-\ominus} \land \\
\text{dom store}' \sqsubset \text{store}' \Leftrightarrow \text{store}' \land \beta' \sqsubset \text{store}' \Leftrightarrow \exists \text{store}' \sqsubset \text{store}' \land \beta' \sqsubset \text{store}'
\end{array}
\]

\square Strengthen Postcondition (B.54)

Using \text{z} |_{0} = \text{z}' |_{0} \land \text{z}^{-\ominus} \land \\
\text{z} \lhd \text{store} = \text{store}' \land \text{post}[\text{z}, \text{z}, \text{store}, \text{store}', \text{store}', \text{store}'][0] \\
\Rightarrow \text{No dereferences of \text{z} in post} \\
\text{z} |_{0} = \text{z}' |_{0} \land \text{z}^{-\ominus} \land \text{post}[\text{z}, \text{z}, \text{store}'] \\
\Rightarrow \\
\text{post}[\text{z}, \text{z}, \text{store}'] \\
\text{and \text{z} |_{0} = \text{z}' |_{0} \land \beta' \sqsubset \beta' |_{0} \Rightarrow \beta' \sqsubset \beta' \sqsubset \text{store}'}

\\\begin{array}{c}
\exists \alpha', \beta'; \text{store}' : \\
\end{array}
\]

\[
\begin{array}{c}
\exists \text{store} \bullet \alpha = \alpha' \land \beta = \beta' \land \text{store} = \text{store}' \land \text{z}^{-\ominus} \land \\
\text{dom store}' \sqsubset \text{store}' \Leftrightarrow \text{store}' \land \beta' \sqsubset \text{store}' \Leftrightarrow \exists \text{store}' \sqsubset \text{store}' \land \beta' \sqsubset \text{store}'
\end{array}
\]

Similarly, there are no dereferences of \text{z} in pre. This observation allows the following
simplification.

\[\equiv\] Contract Frame (B.51)
Weaken Precondition (B.53)
Using \(\bar{z} \not\in store_0 = store' \land \bar{z} = \bar{z}' \land pre[\bar{z} \setminus \bar{z}][store \setminus store_0]\)
\[\Rightarrow\]
\(\bar{z} \not\in store_0 = store' \land \bar{z} = \bar{z}' \land pre[store, \bar{z} \setminus store_0, \bar{z}']\)
\[\Rightarrow\]
\(pre[store, \bar{z} \setminus store', \bar{z}']\)

\[\begin{array}{l}
\left(\text{con } \bar{\alpha}, \bar{\beta}, \text{store}\right) \\
\bar{\alpha}', \text{store}': \begin{array}{c}
\text{pre[store, } \bar{z} \setminus \text{store}', \bar{z}'\text{]} \\
\text{post[ } \bar{z}, \bar{z}_0, \text{store}, \text{store}_0 \setminus \bar{z}', \bar{z}_0, \text{store}', \text{store}_0\text{]} \land \\
\text{dom store}_0 < \text{store}' \overset{\exists \gamma}{\Rightarrow} \text{store}_0 \land \bar{\beta}' \equiv_{\bar{\gamma}'}
\end{array}
\end{array}\]

\[\equiv\] Remove Logical Constant (B.45)
Constrained (8.5)
Ignoring dashed annotations.

\(\bar{\alpha}, \text{store}: \begin{array}{c}
\text{pre} \\
\text{post} \land \emptyset \subseteq \text{store}
\end{array}\)

\[\equiv\] Reference Specification (8.4)
\(\bar{\alpha}: \begin{array}{c}
\text{pre} \land \text{post}
\end{array}\)

QED
Proof of 8.27 from p138 (Inversed reps for Semantic Conversion)

Duplicate (Inversed reps for Semantic Conversion) of 8.27 on page 138.

\[
\text{skip} \equiv \text{rep'; rep'}
\]

Given \( post \) on the mirrored (value) environment \( \overline{z} \), i.e., \( \overline{z} \) and \( \text{store} \) are not free in \( post \), and an initial environment including the references \( \overline{z} \):

\[
\text{(rep; rep') post}
\equiv \text{Definition of AI and DTI}
\equiv (\exists \overline{z}, \text{store} \bullet (\exists \overline{z}', \text{store}' \bullet \text{post}[\text{store}', \overline{z}' \text{store}' \overline{z}'] \land \overline{z}' \equiv \overline{z}'' \land \\
\overline{z} \triangleleft \text{store} = \text{store}'' \land \overline{z}'' \equiv \overline{z} \land \overline{z} \triangleleft \text{store} = \text{store}')
\equiv (\exists \overline{z}, \text{store}, \overline{z}'', \text{store}'' \bullet \text{post}[\text{store}', \overline{z}' \text{store}'' \overline{z}''] \land \overline{z}' \equiv \overline{z}'' \land \\
\overline{z} \triangleleft \text{store} = \text{store}'' \land \overline{z}'' \equiv \overline{z} \land \overline{z} \triangleleft \text{store} = \text{store}' \land \overline{z}'' \equiv \overline{z})
\equiv (\exists \overline{z}, \text{store}, \overline{z}'', \text{store}'' \bullet \text{post}[\text{store}', \overline{z}' \text{store}'' \overline{z}''] \land \overline{z}' \equiv \overline{z}'' \land \\
\overline{z} \triangleleft \text{store} = \text{store}'' \land \overline{z}'' = \overline{z} \land \text{store}'' = \text{store}' \land \overline{z}'' \equiv \overline{z})
\equiv \text{Existential Quantification One Point Rule (B.6)}
\equiv (\exists \overline{z}, \text{store} \bullet \text{post} \land \overline{z} = \overline{z} \land \overline{z} \triangleleft \text{store} = \text{store}' \land \overline{z}'' \equiv \overline{z})
\equiv \text{Choosing store = store' } \cup \{ \overline{z} \mapsto \overline{z}' \}
\equiv (\exists \overline{z} \bullet \text{post} \land \overline{z} \triangleleft (\text{store}' \cup \{ \overline{z} \mapsto \overline{z}' \}) = \text{store}' \land \overline{z}'' \equiv \overline{z})
\equiv \text{post} \land (\exists \overline{z} \bullet \overline{z}'' \equiv \overline{z})
\equiv \text{skip post}
\]

QED
Proof of 8.28 from p138 (Inversed reps for Semantic Conversion B)

Duplicate (Inversed reps for Semantic Conversion B) of 8.28 on page 138.

rep; rep' ⊆ skip

Given post on the mirrored (value) environment (z', store'), i.e., z' and store' are not free in post:

(rep; rep') post
≡ Definition of AI and DTI
(∃ z, store • (∃ z'', store'' • post[store', z''\store'', z''] ∧ z] = z'' ∧
  z ≪ store = store'' ∧ z''⁻) ∧ AI)
≡ (∃ z, store, z'', store'' • post[store', z''\store'', z''] ∧ z] = z'' ∧
  z ≪ store = store'' ∧ z] = z'' ∧ z ≪ store = store' ∧ z''⁻)
≡ Existential Quantification One Point Rule (B.6)
(∃ z, store • post ∧ z] = z'' ∧ z ≪ store = store' ∧ z''⁻)
≡
post ∧ (∃ z, store • z] = z'' ∧ z ≪ store = store' ∧ z''⁻)
⇒
post
≡
skip post
QED
APPENDIX D. PROOFS

Proof of 8.24 from p137 (Transforming Value Semantics Assignments)

Duplicate (Transforming Value Semantics Assignments) of 8.24 on page 137.

For the transformation of value semantics variables \( \tilde{z} \) to reference semantics variables, the assignment

\[
D := E
\]

decodes-as (with respect to \( \tilde{z} \))

\[
(D := E)[\tilde{z} \backslash \tilde{z}']
\]

Proof is straightforward application of Transforming Value Semantics Specification (8.26) using Simple Specification (B.52).

QED

Lemma D.7 (Object Expand Frame) The frame of a specification can be expanded, provided the variables added to the frame are constrained to be object-refinements of their initial value.

\[
\bar{\alpha} : [p , q] \equiv \bar{\alpha}, \bar{\beta} : [p , q \land \bar{\beta}_0 \sqsubseteq \bar{\beta}]
\]

Proof

\[
\bar{\alpha} : [p , q]_z
\]

\[\equiv \]

Specification Statement (B.47)

\[
\lambda pp : Pred \ z \bullet p \land (\forall \bar{\alpha} \bullet q \Rightarrow pp)[\bar{\alpha}_0 \backslash \bar{\alpha}]
\]

\[\Rightarrow \]

Object-Refinement Monotonic Predicate (7.8)

\[
\lambda pp : Pred \ z \bullet p \land (\forall \bar{\alpha} \bullet (\forall \bar{\beta}' \bullet \bar{\beta}_0 \sqsubseteq \bar{\beta}' \Rightarrow (q \Rightarrow pp)[\bar{\beta}' \backslash \bar{\beta}]))[\bar{\alpha}_0 \backslash \bar{\alpha}]
\]

\[\equiv \]

As \( \bar{\beta}_0 \) is not free in \( q \)

\[
\lambda pp : Pred \ z \bullet p \land (\forall \bar{\alpha} \bullet (\forall \bar{\beta}') \bullet \bar{\beta}_0 \sqsubseteq \bar{\beta}' \Rightarrow (q \Rightarrow pp)[\bar{\beta}' \backslash \bar{\beta}]))[\bar{\alpha}_0 \backslash \bar{\alpha}]
\]

\[\equiv \]

Rename the bound.

\[
\lambda pp : Pred \ z \bullet p \land (\forall \bar{\alpha} \bullet (\forall \bar{\beta} \bullet \bar{\beta}_0 \sqsubseteq \bar{\beta} \Rightarrow (q \Rightarrow pp))[\bar{\beta}_0 \backslash \bar{\beta}])][\bar{\alpha}_0 \backslash \bar{\alpha}]
\]

\[\equiv \]

\[
\lambda pp : Pred \ z \bullet p \land (\forall \bar{\alpha} \bullet (\forall \bar{\beta} \bullet \bar{\beta}_0 \sqsubseteq \bar{\beta} \land q \Rightarrow pp))[\bar{\alpha}_0, \bar{\beta}_0 \backslash \bar{\alpha}, \bar{\beta}]
\]

\[\equiv \]

\[
\lambda pp : Pred \ z \bullet p \land (\forall \bar{\alpha} \cdot \bar{\beta} \cdot \bar{\beta}_0 \sqsubseteq \bar{\beta} \land q \Rightarrow pp)[\bar{\alpha}_0, \bar{\beta}_0 \backslash \bar{\alpha}, \bar{\beta}]
\]

\[\equiv \]

Specification Statement (B.47)

\[
\bar{\alpha}, \bar{\beta} : [p , q \land \bar{\beta}_0 \sqsubseteq \bar{\beta}]
\]

QED

Proof of 8.26 from p137 (Transforming Value Semantics Specification)
APPENDIX D. PROOFS

Duplicate (Transforming Value Semantics Specification) of 8.26 on page 137.
For the transformation of value semantics variables $\bar{z}$ (and subset $\bar{\alpha}$) to reference semantics variables, the specification

$$\bar{\alpha}: [pre, post]_*$$

decodes as

$$\bar{\alpha}^\prime: [pre[\bar{z}\setminus\bar{z}]^\prime \land \bar{z}^\prime = \bar{\alpha}, post[\bar{a}_0, \bar{\alpha}\setminus 0, \bar{z}]]_*$$

To aid readability, the proof uses value semantics variables $\bar{z}$ partitioned into $\bar{\alpha}^\prime$ and $\bar{\beta}^\prime$, reference semantics variables $\bar{z}$ partitioned into $\bar{\alpha}$ and $\bar{\beta}$, initial environment variables $\{\bar{z}, store\}$, mirror environment variables $\{\bar{z}, store\}$ and the following abstraction invariant.

$$AI \triangleq \bar{z}\setminus store = \bar{z}^\prime \land \bar{z} = \bar{\alpha} \land \bar{z} \sqsubseteq store = store'$$

and hence $rep\ q \triangleq (\exists \bar{z}, store' \bullet q \land AI)$.

Proof:

$$\bar{\alpha}^\prime: [pre[\bar{z}\setminus\bar{z}]^\prime , post[\bar{z}, \bar{a}_0\setminus\bar{z}, \bar{a}_0']]_*$$

$\equiv$ Reference Specification (8.4)

$$\bar{\alpha}', store': [pre[\bar{z}\setminus\bar{z}]^\prime , post[\bar{z}, \bar{a}_0\setminus\bar{z}, \bar{a}_0'][\bar{\alpha}\setminus\bar{\alpha} \sqsubseteq\bar{\alpha}']$$

$\equiv$ Object Expand Frame (D.7)

$$\bar{z}', store': [pre[\bar{z}\setminus\bar{z}]^\prime , post[\bar{z}, \bar{a}_0\setminus\bar{z}, \bar{a}_0'][\bar{\alpha}\setminus\bar{\alpha} \sqsubseteq\bar{\alpha}' \sqsubseteq\bar{\beta}'$$

$\leq_{rep}$ Data Refine Specification (2.4)

Using $\alpha \equiv \{\bar{z}, store\}$

$\delta \equiv \{\bar{z}, store\}$

$\gamma \equiv \{\}$

$\beta \equiv \{\bar{z}, store\}$

$[[con\ Z', STORE', \bar{z}, store' \bullet$$

$$\bar{z}, store': \begin{cases}
\bar{z}=\bar{z}^\prime \land \bar{z}^\prime = \bar{\alpha} \land \bar{z} \sqsubseteq store = store' \land \\
pre \land \bar{Z}' = \bar{z}^\prime \land \bar{\alpha} = \bar{\alpha}' \land \bar{z} \sqsubseteq store = store' \land \\
(\exists \bar{z}', store' \bullet \bar{z}=\bar{z}^\prime \land \bar{z}^\prime = \bar{\alpha} \land \bar{z} \sqsubseteq store = store' \land \\
(post \land \bar{\alpha} \sqsubseteq store' \land \bar{\beta}' \sqsubseteq \bar{\beta}')[\bar{z}_0, store_0\setminus\bar{Z}', STORE']\end{cases}]
$$

Open window on the specification.

The immediate goals are to remove the existential quantification in the postcondition
and to rearrange the precondition to the precondition in the desired specification.

\[ \equiv \text{Equivalence substitution in precondition on } \text{pre} \text{ using } \overline{z} \models \overline{z}' \]

Split conjunct in postcondition using assumption.

\[ \overline{z}, \text{store}: \]

\[
\overline{z} = \overline{z}' \land \overline{z} = \overline{z}' \land \overline{z} \models \text{store} = \text{store}' \land \\
\text{pre}[\overline{z} \land \overline{z}'] \land \overline{Z}' = \overline{z}' \land \text{STORE}' = \text{store}' \\
(\exists \overline{z}', \text{store}' \bullet \overline{z} = \overline{z}' \land \overline{z} = \overline{z}' \land \overline{z} \models \text{store} = \text{store}' \land \\
(\text{post} \land \varnothing \models \text{store}' \land \overline{\beta}_0 \subseteq \overline{\beta}'[\overline{z}_0, \text{store}' \setminus \overline{Z}', \text{STORE}'])
\]

\[ \equiv \text{Constrained (8.5)} \]

Existential Quantification One Point Rule (B.6)

\[ \overline{z}, \text{store}: \]

\[
\overline{z} = \overline{z}' \land \overline{z} = \overline{z}' \land \\
\overline{z} \models \text{store} = \text{store}' \land \\
\text{pre}[\overline{z} \land \overline{z}'] \land \overline{Z}' = \overline{z}' \land \text{STORE}' = \text{store}' \\
\overline{z} = \overline{z}' \land \overline{z} = \overline{z}' \land \overline{z} \models \text{store} = \text{store}' \land \\
\text{dom} \text{store}'_0 \triangleleft (\overline{z} \models \text{store}) \supseteq \text{store}'_0 \land \\
\beta'_0 \subseteq \beta'[\overline{z}_0, \text{store}'_0, \overline{z} \setminus \overline{Z}', \text{STORE}', \overline{z}]
\]

The current specification allows the aliases \( \overline{z} \) to be altered. This is not, however, needed by the rule that is being proven. Consequently, the frame is contracted.

\[ \equiv \text{Contract Frame (B.51)} \]

Using the knowledge that \( \overline{z}_0 \) is not free in \( \text{post} \)

\[ \overline{z}, \text{store}: \]

\[
\overline{z} = \overline{z}' \land \overline{z} = \overline{z}' \land \overline{z} \models \text{store} = \text{store}' \land \\
\text{pre}[\overline{z} \land \overline{z}'] \land \overline{Z}' = \overline{z}' \land \text{STORE}' = \text{store}' \\
\overline{z} = \overline{z}' \land \overline{z} = \overline{z}' \land \overline{z} \models \text{store} = \text{store}' \land \\
\text{dom} \text{store}'_0 \triangleleft (\overline{z} \models \text{store}) \supseteq \text{store}'_0 \land \\
[\beta'_0 \subseteq \beta'[\overline{z}_0, \text{store}'_0, \overline{z} \setminus \overline{Z}', \text{STORE}', \overline{z}]
\]

\[ \equiv \text{Strengthen Postcondition (B.54)} \]

Using \( \overline{z} = \overline{z}'[\text{store}' \setminus \text{store}_0] \Rightarrow \overline{z} = \overline{z}' \) to eliminate \( \overline{z} = \overline{z}' \)

\[ \overline{z}, \text{store}: \]

\[
\overline{z} = \overline{z}' \land \overline{z} = \overline{z}' \land \overline{z} \models \text{store} = \text{store}' \land \\
\text{pre}[\overline{z} \land \overline{z}'] \land \overline{Z}' = \overline{z}' \land \text{STORE}' = \text{store}' \\
\text{post}[\text{store}', \overline{z}_0, \overline{z}', \overline{z} \models \text{store}, \overline{Z}', \text{STORE}', \overline{z}] \land \\
\text{dom} \text{STORE}' \triangleleft (\overline{z} \models \text{store}) \supseteq \text{STORE}' \land \beta'_0 \subseteq \beta'[\overline{z}_0, \text{store}_0, \overline{z} \setminus \overline{Z}', \text{STORE}]
\]

\[ \equiv \text{Strengthen Postcondition (B.54)} \]

Using \( \overline{z}_0 = \overline{z}' \land \overline{Z}' = \overline{z}' \land \text{STORE}' = \text{store}' \land \overline{z} \models \text{store}_0 = \text{store}' \)

\[ \overline{z}, \text{store}: \]

\[
\overline{z} = \overline{z}' \land \overline{z} = \overline{z}' \land \overline{z} \models \text{store} = \text{store}' \land \\
\text{pre}[\overline{z} \land \overline{z}'] \land \overline{Z}' = \overline{z}' \land \text{STORE}' = \text{store}' \\
\text{post}[\text{store}', \overline{z}_0, \overline{z}', \overline{z} \models \text{store}, \overline{Z}', \text{STORE}', \overline{z}] \land \\
\text{dom}(\overline{z} \models \text{store}_0) \triangleleft (\overline{z} \models \text{store}) \supseteq (\overline{z} \models \text{store}_0) \land \beta'_0 \subseteq \beta'[\overline{z}_0, \text{store}_0, \overline{z} \setminus \overline{Z}', \text{STORE}]
\]
\[\begin{align*}
\square & \text{ Strengthen Postcondition (B.54)} \\
& \text{Using} \\
& \text{store}\preceq\alpha \equiv \exists (\text{dom store}_0 \preceq \text{store}) \exists (\alpha \preceq \text{store}_0) \\
& \equiv \exists (\text{dom store}_0 \preceq \text{store}) \exists (\bar{\beta} \preceq \text{store}_0) \\
& \equiv \text{dom} (\bar{\beta} \preceq \text{store}_0) \preceq (\bar{\beta} \preceq \text{store}) \exists (\bar{\beta} \preceq \text{store}_0) \\
& \end{align*}\]

\[\begin{align*}
\text{store} : & \quad \begin{cases}
\text{post}\text{[store}', \bar{z}, \text{store}\text{'}, \bar{z} \preceq \text{store}, \bar{z} \preceq \text{store}_0, \bar{z}] \land \text{store}\preceq\alpha
\end{cases}
\end{align*}\]

\[\begin{align*}
\square & \text{ Weaken Precondition (B.53)} \\
& \text{Using} \bar{z} = \bar{z} \cup \bar{z}^{-\varnothing} \cup \bar{z} \preceq \text{store} = \text{store}' \land \bar{z} = \bar{z} \cup \text{STORE'} = \text{store}' \land \bar{z}^{-\varnothing} \land \text{pre}\text{[\bar{z}', \bar{z}]} \land \text{post}\text{[\text{store}', \bar{z}_0, \text{store}', \bar{z} \preceq \text{store}, \bar{z} \preceq \text{store}_0, \bar{z}]} \land \text{store}\preceq\alpha
\end{align*}\]

Both \text{pre} and \text{post} only contain dereferences into \text{store}' of variables other than \bar{z}. Consequently, substituting \text{store}' with \text{store} has the same effect as substituting \text{store}' with \bar{z} \preceq \text{store}. A similar argument holds for \text{store}_0.

\[\begin{align*}
\equiv & \quad \text{store} : \begin{cases}
\text{post}\text{[\text{store}', \bar{z}_0, \text{store}_0, \bar{z} \preceq \text{store}, \bar{z} \preceq \text{store}_0, \bar{z}] \land \text{store}\preceq\alpha}
\end{cases}
\end{align*}\]

Ignoring the dash annotations.

\[\begin{align*}
\text{store} : & \quad \begin{cases}
\bar{z}^{-\varnothing} \land \text{pre}\text{[\bar{z}', \text{store}]}, \text{post}\text{[\bar{z}_0, \bar{z} \preceq \text{store}_0, \bar{z}] \land \text{store}\preceq\alpha}
\end{cases}
\end{align*}\]

The substitutions on the postcondition are simplified as \bar{\beta}_0 is not free in \text{post}.

\[\begin{align*}
\equiv & \text{ Reference Specification (8.4)} \\
\alpha : & \quad \begin{cases}
\bar{z}^{-\varnothing} \land \text{pre}\text{[\bar{z}', \text{store}]}, \text{post}\text{[\bar{z}_0, \bar{z} \preceq \text{store}_0, \bar{z}]}
\end{cases}
\end{align*}\]

The proof is completed by closing the window and using Theorem Remove Logical Constant (B.45).

\[\text{QED}\]
D.11 Simultaneous Execution Statement Proofs

Proof of 9.5 from p148 (Introduce Chaotic Simultaneous Execution)

Duplicate (Introduce Chaotic Simultaneous Execution) of 9.5 on page 148.

Any conjunctive code can be refined by simultaneously executing it with $\mathbf{\;\neg\;\wedge\;}:\mathbf{True}$ for variables $\mathbf{\;\neg\;\wedge\;}$ of types $\mathcal{\;\neg\;}W$. For conjunctive $pt$ defined on environment $\mathbf{\;\neg\;}p$, where $\mathbf{\;\neg\;}p$ and $\mathbf{\;\neg\;w}$ are disjoint:

$$pt \sqsupset pt_{\;\neg\;}p_{\;\neg\;}w:\mathbf{True}$$

For conjunctive $pt$ in environment $\mathbf{\;\neg\;}p$:

$$pt \equiv \text{Least Conjunctive Refinement as a Specification (9.3)}$$

$$\mathbf{\;\neg\;}p:\mathbf{\;\neg\;}w:\mathbf{A}(pt), \mathbf{E}(pt)$$

$$\sqsubseteq \text{Open World Specification (4.34)}$$

For $\mathbf{\;\neg\;}w$ outside the environment.

$$\mathbf{\;\neg\;}p, \mathbf{\;\neg\;}w:\mathbf{A}(pt), \mathbf{E}(pt)$$

$$\sqsubseteq \text{Strengthen Postcondition (B.54)}$$

$$\mathbf{\;\neg\;}p, \mathbf{\;\neg\;}w:\mathbf{A}(pt), \mathbf{A}(pt)_{\;\neg\;}p_{\;\neg\;}w_{\;\neg\;}w_{0} \Rightarrow \mathbf{E}(pt)_{\;\neg\;}w_{\;\neg\;}w_{0}$$

$$\equiv \text{Since } \mathbf{\;\neg\;}w \text{ is not free in } pt.$$  

$$\mathbf{\;\neg\;}p, \mathbf{\;\neg\;}w:\mathbf{A}(pt), \mathbf{A}(pt)_{\;\neg\;}p_{\;\neg\;}w_{\;\neg\;}w_{0} \Rightarrow \mathbf{E}(pt)_{\;\neg\;}w_{\;\neg\;}w_{0}$$

$$\equiv \text{Opened Generalised Effect Basic Properties (9.1)}$$

$$\mathbf{\;\neg\;}p, \mathbf{\;\neg\;}w:\mathbf{A}(pt), \mathbf{E}(pt)_{\;\neg\;}p_{\;\neg\;}w_{\;\neg\;}w$$

Heuristically choosing the conjunct. Also using $pt \equiv \mathbf{\;\neg\;}p:\mathbf{A}(pt), \mathbf{E}(pt)$

$$\sqsubseteq \text{Strengthen Postcondition (B.54)}$$

$$\mathbf{\;\neg\;}p, \mathbf{\;\neg\;}w:\mathbf{A}(pt) \land \text{True} , \mathbf{E}(pt)_{\;\neg\;}w \Rightarrow \mathbf{E}(\mathbf{\;\neg\;}w:\mathbf{True} , \mathbf{True} )_{\;\neg\;}p_{\;\neg\;}p$$

Using $True \equiv \mathbf{A}(\mathbf{\;\neg\;}w:\mathbf{True} , \mathbf{True} )$.

$$\equiv \mathbf{\;\neg\;}p, \mathbf{\;\neg\;}w:\mathbf{A}(pt) \land \mathbf{A}(\mathbf{\;\neg\;}w:\mathbf{True} , \mathbf{True} ) , \mathbf{E}(pt)_{\;\neg\;}w \Rightarrow \mathbf{E}(\mathbf{\;\neg\;}w:\mathbf{True} , \mathbf{True} )_{\;\neg\;}p_{\;\neg\;}p$$

$$\equiv \text{Definition Simultaneous Execution}$$

$$pt \sqsupset pt_{\;\neg\;}w\mathbf{True} , \mathbf{True}$$

QED

Proof of 9.6 from p148 (Introduce Object Field for Legacy)
Duplicate (Introduce Object Field for Legacy) of 9.6 on page 148.
When a new field is introduced into an object, the Introduce Chaotic Simultaneous Execution (9.5) rule can be used to enhance the pre-existing methods with the ability to modify the new fields.

Given an object with fields $f_{1..i}$ and conjunctive methods $m_{1..k}$, adding new fields $f_{i+1..i+j}$ (for $j \geq 0$) permits the original methods to be simultaneously executed with chaos on those fields. For new method labels $m_{k+1..k+l}$ (for $l \geq 0$), field values $fv_{1..i+j}$ of types $F_{1..i+j}$, and methods $mv_{1..k+l}$:

\[
\begin{align*}
\text{object} & \\
\text{field } f_{h \in 1..i} : F_h := fv_h \\
\text{method } m_{h \in 1..k} = mv_h \\
\text{end} & \\
\end{align*}
\]

\[
\begin{align*}
\text{object} & \\
\text{field } f_{h \in 1..i+j} : F_h := fv_h \\
\text{method } m_{h \in 1..k} = mv_h & f_{i+1..i+j} f_{i+1..i+j} :: \text{True} \\
\text{method } m_{h \in k+1..k+l} = mv_h & \\
\text{end} & \\
\end{align*}
\]

The proof is a straightforward application of Theorem Introduce Chaotic Simultaneous Execution (9.5) and Definition 7.1.

QED
Appendix E
Symbol Glossary

This appendix provides a glossary and index for the syntax used in this thesis. The page reference provides the definition of the symbol.

# Bag size.
≡ Equality.
∧ Syntactic Definition.
=₀ Equality on type α.
τ(A) This notation provides a shorthand for the type of A.
⊥ The bottom type.
α ≦ β α is a subtype of β.
ℤ The set of integer numbers.
ℝ The set of real numbers.
∅ The empty environment.
E ⊢ f Value f is well formed in environment E.
E ⊣ T Type T is well formed in environment E.
f Value f is well formed in an irrelevant environment.
T Type T is well formed in an irrelevant environment.
A nfi B Variable A is not free in term B.
o :⊥ O The most defined type of object o is O.
false Boolean falsity.
true Boolean truth. p31
∧ Boolean conjunction.
⇒ Boolean implication.
¬ Boolean logical negation.
⇔ Boolean equivalence.
∨ Boolean disjunction.

\( x \odot l \leftarrow b \) Object calculus method update. p39

\( \varsigma(x : X) m \) Object calculus method construction. p36

\([x\backslash v]\) Syntactic substitution of \(x\) with \(v\). p34

\( x \odot l \) Object calculus method invocation. p38

\( \text{Obj} \{ l : B \} \) Object calculus object-types. An introduction is provided in Section 4.1. p35

\( \text{object} \{ l = 6 \} \) Object calculus object. See object calculus object types.

\( \mu(X) \text{Obj} \{ C(X) \} \) Recursive object type. p54

typecase \( a \mid (x : A) d_1 \mid d_2 \) Object calculus dynamic type checking. p54

\( (OT_1 \sqcap OT_2) \) Greatest lower bound of object types. p40

\( (OT_1 \sqcup OT_2) \) Least upper bound of object types. p41

\( (\bigsqcup i \in I \bullet OT_i) \) Generalised greatest lower bound of object types. p41

\( (\bigsqcup i \in I \bullet OT_i) \) Generalised least upper bound of object types. p42

\( \{ l : B \}_{sr} \) The type representing the state which has a variable \(l\) of type \(B\). p42

\( T_{sr} \) The empty state type. p42
APPENDIX E. SYMBOL GLOSSARY

\{l : \mathbb{B}\}_s \quad \text{A state with one variable } l \text{ of type } \mathbb{B} \quad \text{p42}

\alpha \mid_s \{l\} \quad \text{State } \alpha \text{ has identifier } l \text{ cut from it.} \quad \text{p43}

\alpha \cup_s \beta \quad \text{States } \alpha \text{ and } \beta \text{ are combined.} \quad \text{p43}

\text{True} \quad \text{Predicate truth.} \quad \text{p45}

\text{False} \quad \text{Predicate false.} \quad \text{p45}

\Rightarrow \quad \text{Predicate implication.} \quad \text{p45}

\wedge \quad \text{Predicate conjunction.} \quad \text{p45}

\lor \quad \text{Predicate disjunction.} \quad \text{p45}

\Leftrightarrow \quad \text{Predicate equivalence.} \quad \text{p45}

A \sqcap B \quad \text{The demonic choice of statements } A \text{ and } B. \quad \text{p48}

A \sqcup B \quad \text{The angelic choice of statements } A \text{ and } B. \quad \text{p48}

\langle st \rangle \quad \text{Lifts the state transformer } st \text{ to a predicate transformer.} \quad \text{p181}

wp(Prog, post) \quad \text{The weakest precondition function that returns the weakest precondition that establishes } post \text{ after executing } Prog. \quad \text{p10}

[p] \quad \text{Lifts the predicate } p \text{ to a guard.} \quad \text{p180}

\{p\} \quad \text{Lifts the predicate } p \text{ to an assertion.} \quad \text{p179}

[p]_\alpha \quad \text{Validity of predicate } p \text{ for state type } \alpha. \quad \text{p46}

Ptrans \alpha \beta \quad \text{Predicate Transformer type from a postcondition on state } \alpha \text{ to a precondition state } \beta. \quad \text{p46}

p_1 \Rightarrow p_2 \quad \text{Predicate entailment.} \quad \text{p46}

A \sqsubseteq B \quad \text{Statement } A \text{ refines to statement } B. \quad \text{p49}

A \sqsupseteq B \quad \text{Statement } A \text{ is a refinement of statement } B. \quad \text{p49}

A \sqsubsetneq B \quad \text{Statement } A \text{ does not refine to statement } B. \quad \text{p49}

A \preceq_{rep} B \quad \text{Statement } A \text{ data refines to statement } B \text{ under } rep. \quad \text{p14}
APPENDIX E. SYMBOL GLOSSARY

Ref Reference type p69

Ref \{store_i\} A reference type that uses the store \textit{store}_i. p69

\Psi Special function used in definition of semantics for references object specifications to union the ranges of bags. p75

a\dagger The dereference value of reference variable a. p72

\(f::[pre, post]\) An object specification is a specification statement with additional constraints specifically designed for restricting the alteration of object attributes. p66

\textbf{Object}
\begin{align*}
\textbf{field } f & : F \\
\textbf{method } m & \\
\end{align*}
A predicate transformer object-type. p62

\textbf{object}
\begin{align*}
\textbf{field } f & : F := fv \\
\textbf{method } m & = mv \\
\end{align*}
A predicate transformer object. The host object is accessed through self. p62

\(A \sqsubseteq B\) Object \(A\) object-refines to object \(B\). p79

\(A \ll_{rep} B\) An abbreviation used to define object-data-refinement. p110

\(A \prec_{ai} B\) Object \(A\) object-data-refines to object \(B\) under \(ai\). p110

\textbf{class} \textit{classname} \textbf{is}
\begin{align*}
\textbf{field } f & : F := fv \\
\textbf{method } m & = mv \\
\end{align*}
A predicate transformer class. p119

\(a \ll_{\textit{store}}\) \textit{store} is constrained so that all variables except those in \(a\) are object-refinements of their initial values in \textit{store}_0. New locations can also be added. p127

\((a \preceq \overrightarrow{b})\) The variable \(a\) is aliased with those in \(\overrightarrow{b}\), and they are not aliased with those in \(\overrightarrow{c}\). p124

\(a: [pre, post]_{*}\) Reference specifications allow constraints about the store to be left implicit. p126
**APPENDIX E. SYMBOL GLOSSARY**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>$pt_p \odot_q qt$</td>
<td>Miraculous conjunction of $pt$ and $qt$ predicate transformers.</td>
<td>p146</td>
</tr>
<tr>
<td>$pt_p \mid_q qt$</td>
<td>Simultaneous execution of $pt$ and $qt$ predicate transformers.</td>
<td>p144</td>
</tr>
<tr>
<td>$A(pt)$</td>
<td>The generalised assertion of statement $pt$.</td>
<td>p145</td>
</tr>
<tr>
<td>$E_{u \rightarrow v}(pt)$</td>
<td>The generalised effect of statement $pt$.</td>
<td>p145</td>
</tr>
<tr>
<td>$pt \oplus v$</td>
<td>Enlarges statement state space of $pt$.</td>
<td>p144</td>
</tr>
</tbody>
</table>
Appendix F

Glossary

This glossary provides concise definitions and standardisations of terminology used in this thesis. It also acts as an index. The first page reference provides the definition, following which are the pages at which the concept is used in a significant fashion.

Abstract Data Type, 5 An abstract data type is a data structure and a set of operations that are used to access and manipulated the data structure. The operations provided are the only operations that may access and manipulate the data structure. An object is an example of an abstract data type in which the data is encapsulated with its operations.

Aliasing, 121 Aliasing occurs when two or more variables refer to the same entity.

Attributes, 6 The combination of an object’s fields and methods.

Axiomatic Semantics, 28 An axiomatic semantics defines the meaning of a program in an implicit manner. An example is the provision of a programming language semantics via Hoare triples.

Binary Methods, 17 A binary method is a method which has a value parameter and result parameter which is of the same type as that of the host.

Boolean Lattice, 31 A boolean lattice is a lattice in which every element has a unique complement (also in the lattice).

Class, 6 A class is a template from which object instances can be created.

Class-Based, 78 “Class-based languages form the mainstream of object-oriented programming.” e.g., Simula, Smalltalk, C++ [AC96, p11]. The actual properties of class-based languages are controversial. As such, this thesis purports to provide many fundamental features, leaving it to the language designer to include those features deemed appropriate.

Class-based languages are object-oriented languages which have classes.
APPENDIX F. GLOSSARY

Client Enhancement, 106

Cloning, 6 Cloning is the process of copying an object.

Coalescing, 131 The merging of two aliases into one variable.

Complete Lattice, 31 A complete lattice is a lattice which has a top and bottom element.

Construction Monotonicity, 28, 78, 91 The ability to substitute a class with a subclass in a new statement.

Contravariance, 20, 39 See also Covariance. An object’s component type is contravariant if it is permitted to vary anti-monotonically.

Covariance, 20, 27, 39 Covariance is object theory terminology for the monotonicity of a component (type) within an object type with respect to the subtyping relationship.

Data refinement, 11 Data refinement is a relationship between two data types; typically between an abstract, mathematically clear data type and a concrete, implementation-like data type. Simulation is the replacement of the abstract data type with the concrete data type in client code. Typically it is performed to increase the efficiency and/or produce implementable code. This relationship may take the form of a predicate, or as in this thesis, a predicate transformer.

In refinement calculi, when all components of a variable block (or module or class) have been data-refined, the variable block introducing the abstract data type is replaced with one which introduces the concrete data type. The ‘new’, concrete variable block is said to simulate the original. In fact, the concrete variable block is a refinement of the abstract.

Data type, 11 A data type is an abstract data type. In the context of data refinement, the ‘abstract’ is dropped to avoid confusion.

Dynamic Dispatch, 7, 19 The technique used to decide the code to execute for a method invocation of an object.

Fields, 6 Non-invocable, mutable, data attributes of objects.

Host, 6, 64 The host of an attribute (or object) is the object in which the attribute (object) is contained.

Inheritance, 6 Inheritance is the sharing of attributes between a class and its subclass.

Initial Environment, 129

Interference, 147 in the context of aliasing.
Lambda Calculus, 32

Lattice, 31 A partially ordered set in which the meet and join exist for all pairs of elements.

Method, 6 An invocable, procedural portion of an object.

Miraculous Conjunction, 146 The execution of two statements such that the effects of both statements are achieved.

Mirror Environment, 129

Modular Reasoning, 18 The behavioural conformance of polymorphic objects.

Monotonicity, 11

Multiple Dispatch, 19 Multiple dispatch occurs when more than one object (or class) is used to determine the method to execute.

Nested Objects, 80

Object Calculus, 32 Object calculi are used to provide the foundations for object-oriented language in a fashion analogous to manner in which $\lambda$-calculi provide the foundations for procedural languages.

Object Identity, 121

Object-Based, 78 “Object-based languages simplify and generalise class-based languages by reducing classes to more primitive notions.” [AC96, p1]

Object-Data-Refinement, 106

Object-Refinement Algorithmic, 79

Operational Semantics, 28 Operational semantics are based on implementable models, e.g., stacks and stores (also known as closures, stores are functions mapping locations to values). They provide a machine oriented description of how the language works.

Polymorphism, 6 Typically used to refer to subtyping polymorphism which is the use of an object as an instance of a supertype.

Primary Variable, 131 When two variables are coalesced, the one remaining is termed the primary.

Product Types, 23 Product types are better known under the name Cartesian products.
Prototype, 6 A prototype is the analogy of a class in an object-based language. A prototype is an object used as a template for cloning.

Reference Assignment, 90

Reference cloning, 91 The copying of an object in a semantics for references.

Reference Specification, 123 An enhanced specification that implicitly encapsulates the store constraints in a semantics for references.

Simulation, 11, 16 see Data refinement. In the context of data refinement, simulation, (actually a special case refinement) is the replacement of an abstract data type by its data refinement in client code.

The process starts with two data types where it has been shown that they exist in a data refinement relationship. Simulation is the proof of refinement between the abstract client code and the concrete client code.

The behaviour of the abstract client code has been ‘simulated’ by the concrete client code—even though they work on different state spaces.

Store, 121 A store is a function mapping references (pointers to objects) to their values.

Subclassing, 6 The word ‘subclassing’ has various meanings depending upon its context. In the literature review, its definition is provided within the context it is used. For class based refinement as in Section 7.5, a subclass denotes a class refinement that is also a subtype. Hence subclass instances can be used in place of the superclass instances.

Subsumption, 7 The recognition of an object as an instance of a supertype of its own type.

Sum Types, 23 Sum types are the composition of multiple distinct base types. The base types are projected into the summation. No two projected elements of the base types map to the same element within the summation.

Upwards Closed, 83